

# Curso Básico de JDBC.





# Acerca de:

En la compilación de esta obra se utilizaron libros conocidos en el ambiente Java, gráficas, esquemas, figuras de sitios de internet, conocimiento adquirido en los cursos oficiales de la tecnología Java. En ningún momento se intenta violar los derechos de autor tomando en cuenta que el conocimiento es universal y por lo tanto se puede desarrollar una idea a partir de otra.

La intención de publicar este material en la red es compartir el esfuerzo realizado y que otras personas puedan usar y tomar como base el material aquí presentado para crear y desarrollar un material mucho más completo que pueda servir para divulgar el conocimiento.



# CONTENIDO

Modulo 01: Introducción al JDBC.

Modulo 02: Práctica de Laboratorio.

Modulo 03: JDBC Avanzado.

Modulo 04: Práctica de Laboratorio.



## Modulo 01 Introducción al JDBC.



# Objetivos.

- Después de completar este modulo, el participante será capaz de:
  - Conocer las características principales de la tecnología Java.
  - Explicar los pasos a seguir para conectarse a una Base de Datos (BD).
  - Conocer los métodos principales de la API de JDBC.
  - Identificar la manipulación de las sentencias SQL mediante programas Java.
  - Identificar el manejo de las excepciones con JDBC.



# Contenido.

- Introducción.
- Evolución de JDBC.
- Arquitectura del API JDBC.
- Descripción general del API JDBC.
- Drivers JDBC.
- ¿Qué necesitamos para trabajar con JDBC?.
- Conexión de la Base de Datos.
- Conexión a JDBC.
- Cargar el controlador JDBC.
- Establecer una conexión JDBC.
- TestConexion.java
- Creación de sentencias.



# Contenido.

- Objetos Statement.
- Sentencia executeUpdate().
- Insertar.java
- Sentencia executeQuery()
- Consultar.java
- Tipos de Datos y Conversiones.
- ResultSetMetaData.
- TestResultSetMetaData.java
- DatabaseMetaData.
- TestMetaData.java



# Introducción.

- JDBC es un conjunto de clases e interfaces Java que permiten la manipulación de sentencias SQL de una fuente de datos (base de datos).
- La interface Java (API de JDBC) proporciona a las aplicaciones Java un mecanismo estándar e independiente de la plataforma para el acceso a la mayoría de las bases de datos existentes.
- La API de JDBC define un conjunto de clases e interfaces que proporcionan toda la funcionalidad que el acceso a base de datos requiere, tal como la ejecución de consultas SQL o el tratamiento de los resultados.
- Cada fabricante de base de datos se encargará de proporcionar un driver JDBC específico para su base de datos.



# Introducción.

- Las actividades básicas de programación que vamos a utilizar en JDBC:

- 1 Conectarse a una fuente de datos, como una base de datos.
- 2 Enviar Querys y Updates a la base de datos.
- 3 Recuperar y procesar los resultados obtenidos de la base de datos en respuesta al Query obtenido.

Todo el conjunto de clases e interfaces que constituyen JDBC se encuentran dentro del paquete `java.sql` principalmente pero también existe el paquete `javax.sql`.



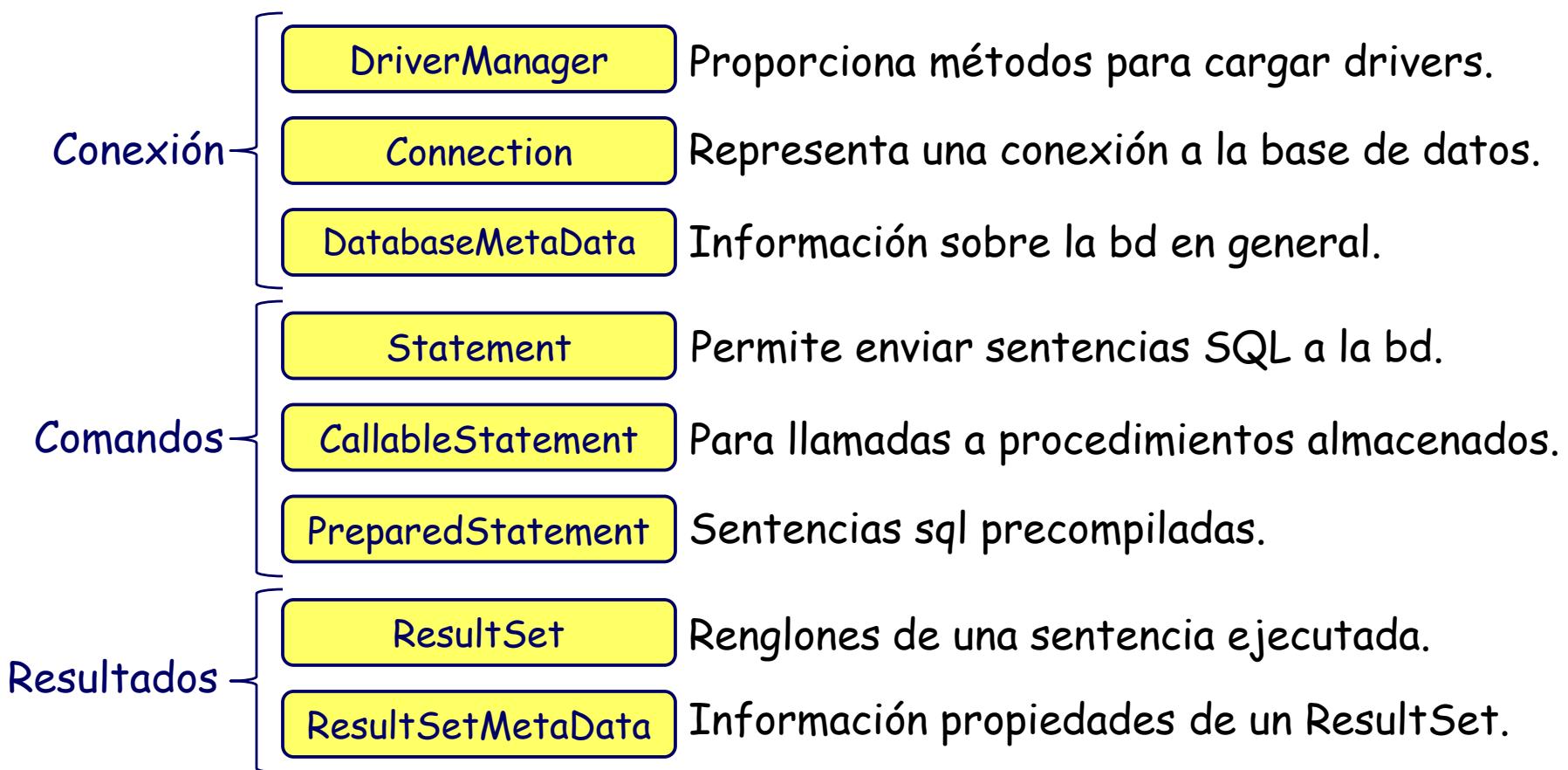
# Componentes JDBC.

- El producto JDBC incluye 4 componentes:
  - 1 El API de JDBC Java™ Standard Edition (Java™ SE) Java™ Enterprise Edition (Java™ EE) 
  - 2 JDBC Driver Manager Conecta aplicaciones Java con el Driver Correcto de JDBC. Se puede realizar por conexión directa o vía DataSource.
  - 3 JDBC Test Suite Comprueba si un Driver cumple con los requisitos JDBC.
  - 4 Puente JDBC-ODBC Permite que se puedan utilizar los Drivers ODBC como si fueran de tipo JDBC.



# Descripción general del API JDBC.

- Las interfaces principales de la API JDBC que se encuentran en la librería java.sql:



# Tipos de drivers JDBC.

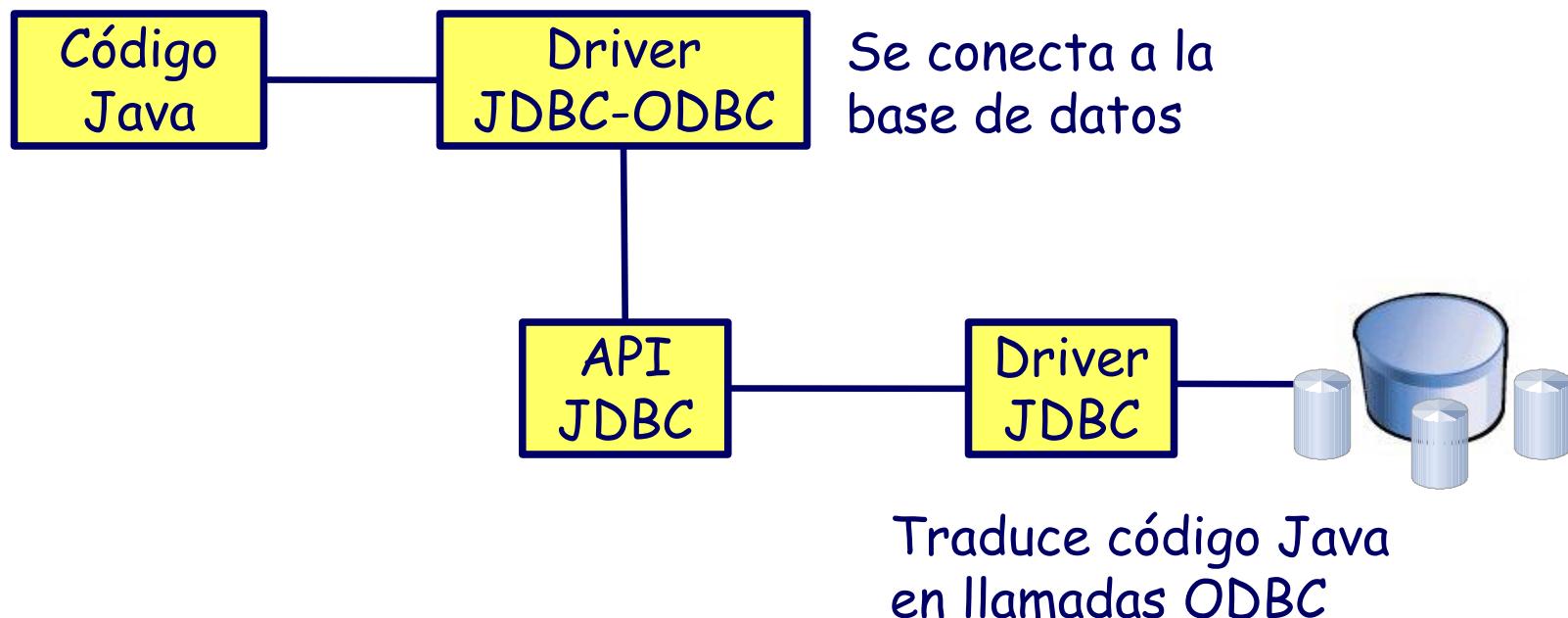
- Un driver JDBC es una implementación de varias interfaces especificadas en los paquetes `java.sql` y `javax.sql`.
- Es una capa de software intermedio que traduce las llamadas JDBC a las APIs específicas de cada vendedor.
- Existen cuatro tipos de controladores JDBC, cada uno numerado del 1 al 4 en orden creciente en relación a la independencia de la plataforma, desempeño, etc.

- › Driver tipo 1: utilizan una API nativa estándar.
- › Driver tipo 2: utilizan una API nativa de la base de datos.
- › Driver tipo 3: utiliza un servidor remoto con una API genérica.
- › Driver tipo 4: es el método más eficiente de acceso a base de datos.



# Driver Tipo 1: JDBC-ODBC.

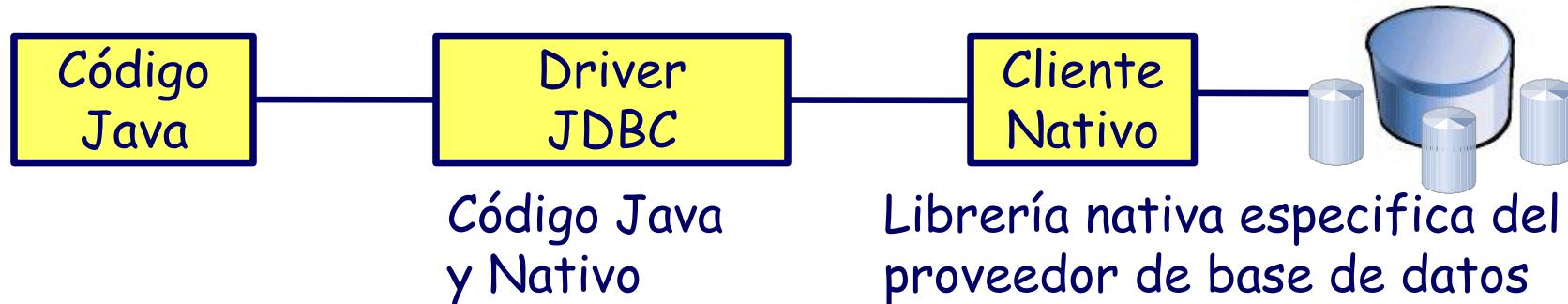
- El driver JDBC-ODBC es parte de la plataforma Java. No es un driver 100% Java.
- Traduce las llamadas a JDBC a invocaciones ODBC a través de librerías ODBC del sistema operativo.



## Driver Tipo 2: Native API partly-java.

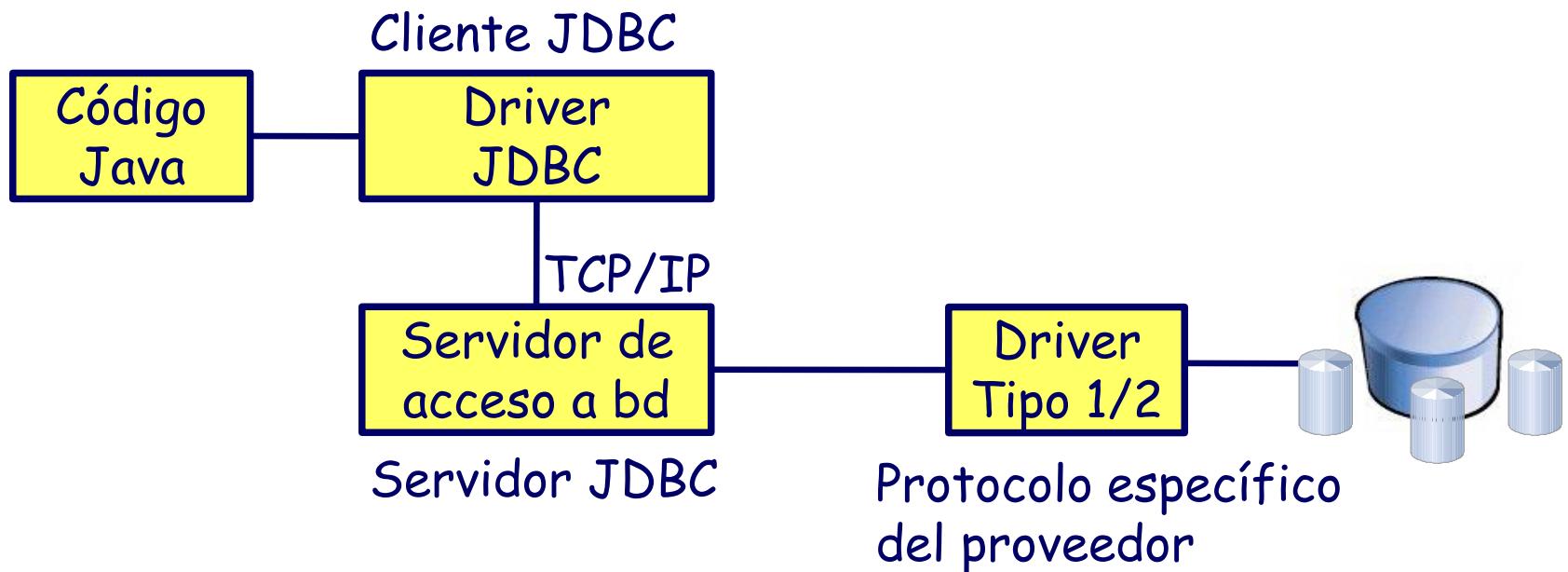
- Son drivers escritos parte en Java y parte en código nativo. El driver usa una librería cliente nativa, específica de la base de datos con la que quiere conectarse. No es un driver 100% Java.
- La aplicación Java hace una llamada a la base de datos a través del driver JDBC, el driver traduce la petición, en invocaciones a la API del fabricante de la base de datos.

Es un driver que usa protocolos de acceso  
a datos optimizados por el fabricante



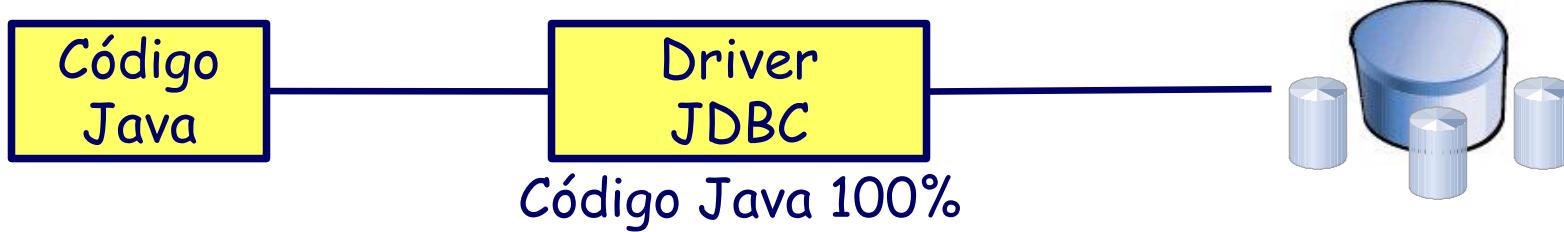
# Driver Tipo 3: JDBC-Net pure Java.

- Son drivers que usan un cliente Java puro (cliente JDBC) que se comunica con un middleware server (servidor JDBC) usando un protocolo independiente de la base de datos (TCP/IP).
- Convierte las llamadas en un protocolo (por ejemplo TCP/IP) que puede utilizarse para interactuar con la base de datos.

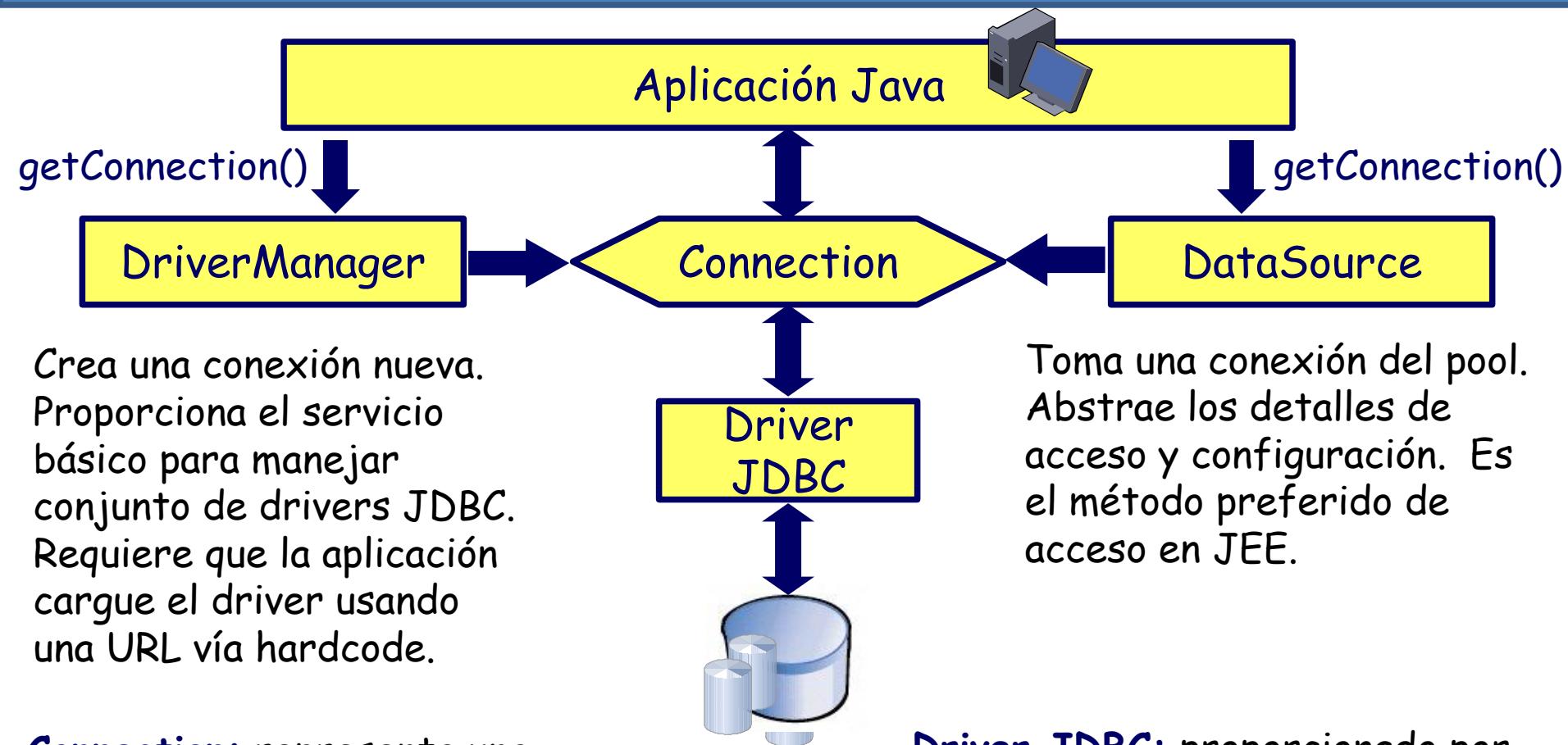


# Driver Tipo 4: Native-Protocol Pure Java.

- Son drivers suministrados por el fabricante de la base de datos y su finalidad es convertir llamadas JDBC en un protocolo de red (usando sockets) comprendido por la base de datos.
- Es el método más eficiente de acceso a base de datos.



# Uso de JDBC en aplicaciones Java.



# Pasos para utilizar JDBC en aplicaciones Java.

```
Class.forName("driver");
```

```
DriverManager.getConnection(url,usr,pwd);
```

```
conn.createStatement();  
conn.prepareStatement(sql);  
conn.prepareCall(sql);
```

```
stmt.executeQuery();  
stmt.executeUpdate();
```

(create, alter, drop) DDL

(insert, update, delete) DML

```
stmt.close();  
conn.close();
```

Cargar/Registrar  
el Driver JDBC

1

Obtener la  
conexión

2

Crear el comando  
SQL

3

Ejecutar el  
comando SQL

4

Procesar los  
resultados

4.1

Liberar recursos

5



# Cargar el driver JDBC.

- Para conectarnos a una base de datos a través de JDBC desde una aplicación Java, lo primero que necesitamos es cargar el driver.
- La sintaxis para cargar el driver es:

```
Class.forName("NombreDelDriver");
```

Facilitado por  
el fabricante

- Por ejemplo, si vamos a emplear el driver el puente JDBC-ODBC para conectarnos a una base de datos MS Access, entonces el código sería:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver" );
```



# Cargar el driver JDBC.

- En nuestro curso utilizaremos el driver nativo de MySQLConektor/J para acceder a MySQL. Por lo tanto para cargar el driver tendríamos que escribir en nuestro código Java:

```
Class.forName(" com.mysql.jdbc.Driver " );
```

- Llamando a Class.forName la aplicación Java carga el driver JDBC y entonces ya nos podemos conectar con la base de datos invocando el método DriverManager.getConnection.



# Obtener la conexión.

- Para conectarnos a la base de datos una vez cargado el driver, utilizaremos el método getConnection que a su vez define una URL que indicará la ubicación de la base de datos:

```
Connection conn = DriverManager.getConnection(url,usr,pwd);
```

Es el protocolo

Identifica el driver de la bd

Indica el nombre y en donde se encuentra la BD

Usuario para acceder a la base de datos

Clave de acceso

jdbc:<subprotocol>:<subnombre>



# Obtener la conexión.

- Por ejemplo para establecer una conexión con MS Access podríamos escribir:

```
Connection conn;  
String url = "jdbc:objc:NombreBaseDatos";  
conn = DriverManager.getConnection(url, "", "");
```

- Y para obtener la conexión con la base de datos MySQL:

```
Connection conn;  
String url = "jdbc:mysql://localhost/NombreBaseDatos";  
conn = DriverManager.getConnection(url, "", "");
```



# Obtener la conexión.

Ejemplo:

```
package cursojdbc;  
import java.sql.*;
```

```
public class TestConexion {
```

```
static String bd = "MI_BIBLIOTECA";  
static String login = "root";  
static String password = "admin";  
static String url = "jdbc:mysql://localhost/" + bd;
```

```
public static void main(String[] args) {  
    try {
```

```
        Class.forName("com.mysql.jdbc.Driver");  
        Connection conn =  
            DriverManager.getConnection(url, login, password);
```

Los programas deben declarar el uso de este paquete.

Constantes de configuración para el acceso a la bd vía hardcode.

Carga el driver en memoria e intenta realizar la conexión a la bd.



# Obtener la conexión.

Ejemplo:

```
if (conn != null) {  
    System.out.println("Conexión a la bd " + url + "....ok!!");  
    conn.close();  
}  
} catch (ClassNotFoundException cnfe) {  
    System.out.println("Driver JDBC no encontrado");  
    cnfe.printStackTrace();  
} catch (SQLException sqle) {  
    System.out.println("Error al conectarse a la BD");  
    sqle.printStackTrace();  
} catch (Exception e) {  
    System.out.println("Error general");  
    e.printStackTrace();  
}  
}
```

Si se logra la conexión se despliega un mensaje de usuario. En caso contrario lanza una excepción.

2/2



# Crear el comando SQL.

- Ya que hemos establecido una conexión con la base de datos usando el método getConnection de DriverManager, ahora podemos crear sentencias SQL utilizando la interface Statement que provee métodos para realizar esas tareas. Entonces tenemos que escribir en nuestro código Java:

```
Statement stmt = conn.createStatement();
```

- Ahora para que podamos utilizar las sentencias UPDATE, INSERT, DELETE,SELECT tenemos que utilizar los métodos:

**executeUpdate**

Retorna un número entero indicando la cantidad de registros afectados (UPDATE, INSERT,DELETE).

**executeQuery**

Regresa un conjunto de resultados que se almacenan en un objeto ResultSet.



# Ejecutar el comando SQL: executeUpdate.

- Utilizamos el método executeUpdate para sentencias SQL de tipo DML que crean, modifican o eliminan datos de las tablas, también lo podemos usar en sentencias de tipo DDL tales como crear, modificar, borrar tablas.
- En general, vamos a utilizar executeUpdate para todo aquello que no regrese un conjunto de resultados. Por ejemplo:

```
String cadSQL = (" INSERT INTO autor " +  
    " VALUES(127,'Raúl','Oramas' ");
```

Observa el uso  
de las comillas  
simples.

La instrucción SQL debe ser sintácticamente correcta.

```
Statement stmt = conn.createStatement();  
int r = stmt.executeUpdate(cadSQL);
```

El valor de retorno se guarda en una  
variable entera.

Se crea a partir del  
objeto Connection.



# Ejecutar el comando SQL: executeUpdate.

Ejemplo:

```
package cursojdbc;
import java.sql.*;
public class TestExecuteUpdate {
    final static String bd = "MI_BIBLIOTECA";
    final static String login = "root";
    final static String password = "admin";
    final static String url = "jdbc:mysql://localhost/" + bd;
    Connection conn;
    Statement stmt;

    public TestExecuteUpdate() throws SQLException,
        ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(url, login, password);
        stmt = conn.createStatement();
    }
}
```



# Ejecutar el comando SQL: executeUpdate.

Ejemplo:

```
public void operacionesBD() throws SQLException {  
    int r = 0;  
    String cadSQL = null;  
  
    cadSQL = "INSERT INTO autor VALUES(127,'Peter','Norton')";  
    r = stmt.executeUpdate(cadSQL);  
    System.out.println(r + " registro agregado.");  
    cadSQL = "INSERT INTO autor VALUES(128,'Laura','Lemay')";  
    r = stmt.executeUpdate(cadSQL);  
    System.out.println(r + " registro agregado.");  
  
    cadSQL = "UPDATE autor SET nombreAutor='Pedro' WHERE  
              nombreAutor='Peter'";  
    r = stmt.executeUpdate(cadSQL);  
    System.out.println(r + " registro modificado.");
```

1/2



# Ejecutar el comando SQL: executeUpdate.

Ejemplo:

```
cadSQL = "DELETE FROM autor WHERE nombreAutor='Pedro'";
r = stmt.executeUpdate(cadSQL);
System.out.println(r + " registro eliminado.");
cadSQL = "DELETE FROM autor WHERE nombreAutor='Laura'";
r = stmt.executeUpdate(cadSQL);
System.out.println(r + " registro eliminado.");
stmt.close();
conn.close();
}
public static void main(String[] args) {
    try {
        TestExecuteUpdate test = new TestExecuteUpdate();
        test.operacionesBD();
    } catch (SQLException sqle) {
        System.out.println(sqle);
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

1/2

30



# Ejecutar el comando SQL: executeQuery.

- Cuando se ejecutan sentencias SELECT usando el método executeQuery, se obtiene como respuesta un conjunto de resultados, que en Java es representado por un objeto ResultSet

`ResultSet rs = stmt.executeQuery("SELECT * FROM autor");`

Regresa los resultados  
en el objeto ResultSet.



AutorID	Nombre	Apellido
1	Harvey	Deitel
2	Paul	Deitel
3	Jackie	Barker
4	Paul	Sanghera
5	Ivor	Horntons
6	Todd	M.Thomas



## Ejecutar el comando SQL: executeQuery.

- El objeto ResultSet controla la recuperación de los registros.
- Representa un cursor (iterador) sobre los resultados:
  - Movimiento: métodos next() y previous().
  - Inicialmente el cursor está posicionado antes del primer registro.
- Depende del objeto consulta: cada vez que se realice una consulta se pierden los resultados.



## Ejecutar el comando: executeQuery.

- Tenemos dos alternativas para acceder a las columnas del resultado:

```
rs.getString("nombre"); //nombre de la columna  
rs.getString(1); //posición en la consulta
```

- El acceso por posición es útil cuando:
  - Acceso a una columna derivada, por ejemplo, calcular la media.
  - Cuando hay columnas con los mismos nombres (join)
- Recuperación de los valores de las columnas:
  - Métodos de acceso (getXXX)



# Ejecutar el comando SQL: executeQuery.

Ejemplo:

```
package cursojdbc;
import java.sql.*;
public class TestExecuteQuery {
    final static String bd = "MI_BIBLIOTECA";
    final static String login = "root";
    final static String password = "admin";
    final static String url = "jdbc:mysql://localhost/" + bd;
    Connection conn;
    Statement stmt;

    public TestExecuteQuery() throws SQLException,
                                   ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(url, login, password);
        stmt = conn.createStatement();
    }
}
```

1/2



# Ejecutar el comando SQL: executeQuery.

Ejemplo:

```
public void operacionesBD() throws SQLException {  
    ResultSet rs;  
    String cadSQL = null;  
  
    cadSQL = "SELECT * FROM autor";  
    rs = stmt.executeQuery(cadSQL);  
  
    while(rs.next()) {  
        System.out.print("Autor ID: " + rs.getString(1));  
        System.out.print("\tNombre: " + rs.getString(2));  
        System.out.print("\t" + rs.getString(3) + "\n");  
    }  
    rs.close();  
    stmt.close();  
    conn.close();  
}
```

Obtiene un objeto ResultSet a partir de una cadena SQL.

Navega a través del ResultSet utilizando el método next

Utilizando los métodos getXXX obtiene los valores de las columnas en el ResultSet. Se obtiene por el nombre de las columnas o por su posición.



# Ejecutar el comando SQL: executeQuery.

Ejemplo:

```
public static void main(String[] args) {  
    try {  
        TestExecuteQuery test = new TestExecuteQuery();  
        test.operacionesBD();  
    } catch (SQLException sqle) {  
        System.out.println(sqle);  
    } catch (Exception e) {  
        System.out.println(e);  
    }  
} //fin main  
  
}
```

1/2



# Tipos de datos y conversiones.

- Cuando se lanza un método getXXX sobre un objeto ResultSet, el driver JDBC convierte el dato que se quiere recuperar a el tipo Java especificado y entonces devuelve un valor Java adecuado
- La conversión de tipos se puede realizar gracias a la clase java.sql.Types. En esta clase se definen lo que se denominan tipos de datos JDBC, que se corresponde con los tipos de datos SQL estándar



# Tipos de datos y conversiones.

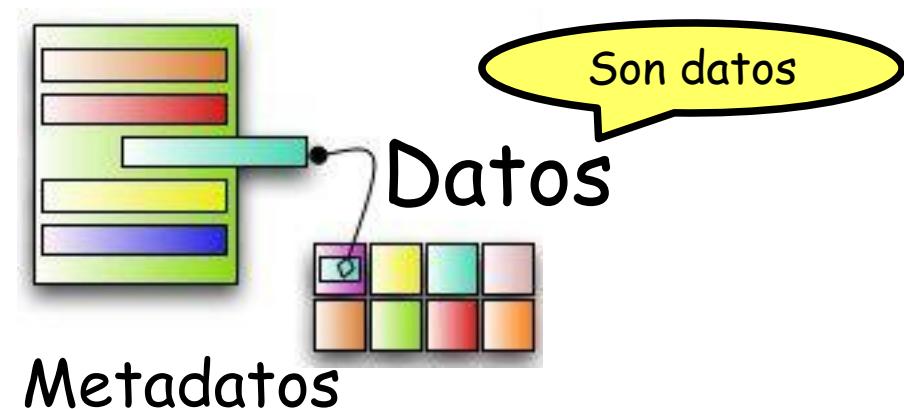
- El mapeo o conversión es la siguiente:

Tipos JDBC	Tipos Java	Tipos JDBC	Tipos Java
BIGINT	getLong()	LONGVARCHAR	getString()
BINARY	getBytes()	NUMERIC	getBigDecimal()
BIT	getBoolean()	OTHER	getObject()
CHAR	getString()	REAL	getFloat()
DATE	getDate()	SMALLINT	getShort()
DECIMAL	getBigDecimal()	TIME	getTime()
DOUBLE	getDouble()	DATETIME	getTimeStamp()
FLOAT	getDouble()	TINYINT	getByte()
INTEGER	getInt()	VARBINARY	getBytes()
LONGVARBINARY	getBytes()	VARCHAR	getString()



# Trabajando con metadatos.

- Los metadatos son datos acerca de los datos. Datos que explican la naturaleza de otros datos.
- Con el acceso a los metadatos podemos conocer la estructura de la base de datos (no su contenido) y nos permite desarrollar aplicaciones independientemente del esquema de la base de datos.



# ResultSetMetaData.

- Los métodos de ResultSetMetaData nos permite determinar las características de un objeto ResultSet.
- Por ejemplo podemos determinar:

- › El número de columnas.
- › Información sobre una columna, tal como el tipo de datos, la longitud, la precisión y la posibilidad de contener nulos.
- › La indicación de si una columna es de solo lectura, etc.

Conjunto de datos  
Obtenido en un ResultSet



```
ResultSetMetaData rsmd = rs.getMetaData();
```



# ResultSetMetaData.

Ejemplo:

```
package cursojdbc;
import java.sql.*;
public class MetaData01 {
    final static String bd = "MI_BIBLIOTECA";
    final static String login = "root";
    final static String password = "admin";
    final static String url = "jdbc:mysql://localhost/" + bd;
    Connection conn;
    Statement stmt;

    public MetaData01() throws SQLException,
                           ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(url, login, password);
        stmt = conn.createStatement();
    }
}
```

1/2



# ResultSetMetaData.

Ejemplo:

```
public void desplegarMetaDatosResultSet() throws SQLException {
    ResultSet rs = stmt.executeQuery("SELECT * FROM AUTOR");
    ResultSetMetaData rsmd = rs.getMetaData();

    int nColumnas = rsmd.getColumnCount();
    System.out.println("Nombre tabla: " + rsmd.getTableName(1));
    System.out.println("La tabla tiene : " + rsmd.getColumnCount() +
columnas.);

    // obtiene los nombres de las columnas y el tipo de dato SQL
    asociado
    for (int i = 1; i < nColumnas + 1; i++) {
        System.out.println("Columna: " + rsmd.getColumnName(i)
        + "\t" + rsmd.getColumnTypeName(i) +
        "\t" + rsmd.getPrecision(i));
    }
    rs.close();
    stmt.close();
```

1/2

42



# DatabaseMetaData.

- La interface DatabaseMetaData contiene más de 150 métodos para recuperar información de un Base de Datos (catálogos, esquemas, tablas, tipos de tablas, columnas de las tablas, procedimientos almacenados, vistas etc.) así como información sobre algunas características del controlador JDBC que estemos utilizando
- Estos métodos son útiles cuando se escribe aplicaciones genéricas que pueden acceder a diversas Bases de Datos.

```
DatabaseMetaData dbmd = conn.getMetaData();
```



# DatabaseMetaData.

Ejemplo:

```
package cursojdbc;
import java.sql.*;
public class DBMetaData01 {
    final static String bd = "MI_BIBLIOTECA";
    final static String login = "root";
    final static String password = "admin";
    final static String url = "jdbc:mysql://localhost/" + bd;
    Connection conn;
    Statement stmt;

    public DBMetaData01() throws SQLException,
                               ClassNotFoundException {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(url, login, password);
        stmt = conn.createStatement();
    }
}
```

1/2



# DatabaseMetaData.

Ejemplo:

```
public void desplegarMetaDatosBD() throws SQLException {  
  
    DatabaseMetaData dbmd = conn.getMetaData();  
  
    System.out.println("URL de la BD: " + dbmd.getURL());  
    System.out.println("Usuario de la BD: " + dbmd.getUserName());  
    System.out.println("Nombre del driver de la BD: " +  
dbmd.getDriverName());  
    ResultSet tablas = dbmd.getTables(null,null,null,null);  
    System.out.print("Tablas de la BD: ");  
    while (tablas.next()) {  
        System.out.print(tablas.getString("TABLE_NAME") + "\t");  
    }  
    stmt.close();  
    conn.close();  
}
```



# DatabaseMetaData.

Ejemplo:

```
public static void main(String[] args) {  
    try {  
        DBMetaData01 test = new DBMetaData01();  
        test.desplegarMetaDatosBD();  
    }  
    catch(SQLException sqle) {  
        System.out.println(sqle);  
    }  
    catch(Exception e) {  
        System.out.println(e);  
    }  
} //fin main  
  
}
```

1/2



# Modulo 02

# Práctica de Laboratorio.



# Objetivos.

- Después de completar este modulo, el participante será capaz de:
  - Conectarse a una base de datos a través de JDBC
  - Utilizar las instrucciones de JDBC para manipular las tablas de una base de datos.



# Práctica de Laboratorio.

- **Paso 1.** Crear una base de datos denominada ControlEscolar y crear cinco tablas en base a las definiciones siguientes :

**Curso**

CursoId	INT
NombreCurso	CHAR 20

**Instructor**

InstructorId	INT
NombreInstructor	CHAR 60

**CursoInstructor**

CursoId	INT
InstructorId	INT

**Estudiante**

EstudianteId	INT
NombreEstudiante	CHAR 60

**CursoEstudiante**

CursoId	INT
EstudianteId	INT



# Práctica de Laboratorio.

- **Paso 2.** Crear una clase JDBC`Lab01` que contenga los métodos para:
  - a) Insertar datos en cada una de las tablas
  - b) Mostrar los datos del instructor y los cursos que esta impartiendo
  - c) Actualizar el nombre de un instructor/estudiante/curso
  - d) Desplegar los datos de los estudiantes registrados
  - e) Dado un número de curso, desplegar el nombre del curso, y el nombre de los estudiantes registrados en ese curso
- **Paso 3.** Crear una clase JDBC`Lab01Tester` para demostrar el funcionamiento de JDBC`Lab01`



## Modulo 03 JDBC Avanzado.



# Objetivos.

- Después de completar este modulo, el participante será capaz de:
  - Describir el uso de las clases heredadas de la clase Statement.
  - Discutir acerca de los procedimientos almacenados.
  - Discutir sobre el manejo de transacciones, cursores y procesamiento de múltiples resultados.



# Contenido.

- Introducción.
- La clase PreparedStatement.
- TestPreparedStatement.java
- La clase CallableStatement.
- Transacciones.
- Niveles de aislamiento transaccional.
- Excepciones JDBC.



# Introducción.

- La especificación JDBC provee dos clases para la programación sofisticada en las bases de datos: PreparedStatement y CallableStatement.
- Con PreparedStatement podemos ejecutar instrucciones SQL precompiladas y
- CallableStatement permite ejecutar los procedimientos almacenados de las Bases de Datos.
- A continuación estudiaremos cada una de estas instrucciones.



# La clase PreparedStatement.

- El problema con Statement sucede cuando la consulta se realiza dentro de un ciclo y varía sólo en unos valores:

```
stmt.executeQuery("SELECT * FROM Cliente WHERE codigo = " + i);
```

- La base de datos planifica cada consulta.
- Conviene disponer de una consulta con parámetros PreparedStatement que es una especialización de Statement que permite definir consultas parametrizadas.
- La BD sólo planifica la consulta cuando se crea.
- Evitan tener que formatear los datos al construir la cadena de consulta: '' para cadenas, fechas y horas.



# La clase PreparedStatement.

- También se crean a partir de la conexión:

```
PreparedStatement pstmt =  
conexion.prepareStatement("SELECT * FROM Cliente  
WHERE codigo = ?")
```

- Los parámetros de entrada se especifican por posición utilizando métodos setXXX: psmt.setInt(1, 20);
- Misma equivalencia que los getXXX de ResultSet.
- Los valores se conservan entre ejecuciones.
- Borrar parámetros: clearParameters()
- Ejecución:
  - Consulta: executeQuery().
  - Actualización: executeUpdate().



# TestPreparedStatement.java

```
Statement ipStmt = conn.prepareStatement(  
"INSERT INTO contactos(nombre,telefono,email)  
VALUES(?, ?, ?)"),  
ipStmt.setString(1, "Juan");  
ipStmt.setString(2, "01667-7505816");  
ipStmt.setString(3, "juan@hotmail.com");  
int i = ipStmt.executeUpdate();  
  
void setBoolean(int paramIndex, boolean x)  
void setDate(int paramIndex, Date x)  
void setDouble(int paramIndex, double x)  
void setFloat(int paramIndex, float x)  
void setInt(int paramIndex, int x)  
void setLong(int paramIndex, long x)  
void setNull(int paramIndex, int x)  
void setString(int paramIndex, String x)
```

Parámetros IN



# La clase CallableStatement.

- CallableStatement es el modo estándar de llamar procedimientos almacenados con la sintaxis de escape SQL de procedimiento almacenado de API JDBC.
- La sintaxis de escape SQL soporta dos formas de procedimientos almacenados. La primera forma incluye un parámetro de resultado conocido como parámetro OUT, y la segunda forma no utiliza parámetros OUT. Cada una de las formas puede tener parámetros IN.
- Cuando el controlador JDBC encuentra “{call PROC\_ALM}”, traducirá esta sintaxis de escape al SQL nativo utilizado en la Base de Datos.



# La clase CallableStatement.

- La sintaxis típica para llamar a un procedimiento almacenado es:

```
{call nombre_procedimiento[?, ?, ...?]}
```

```
{? = call nombre_procedimiento[?, ?, ...?]}
```

- Los objetos CallableStatement son creados con el método prepareCall de la interface Connection:

```
Connection conn = DriverManager.getConnection();
CallableStatement sp = conn.prepareCall("{call sp(?, ?)}");
```



# Transacciones.

- Ejecución de bloques de consultas SQL manteniendo las propiedades ACID (Atomicity-Consistency-Isolation-Durability), es decir permite garantizar integridad ante fallas y concurrencia de transacciones
  - Atomicity: Las operaciones en ella incluida deben ser realizadas todas en grupo o ninguna.
  - Consistency: La Base de Datos ha de quedar en un estado que no viole la integridad de la misma.
  - Isolation: La lógica debe poder proceder.
  - Duradero: Si tiene éxito la transacción, las operaciones serán o pasarán a un estado persistente en la Base de Datos.



# Transacciones.

- Una transacción que termina exitosamente se compromete (commit).
- Una transacción que no termina exitosamente se aborta (rollback).
- En JDBC por omisión cada sentencia SQL se compromete tan pronto se ejecuta, es decir una conexión funciona por defecto en modo auto commit.
- Para ejecutar varias sentencias SQL en una misma transacción es preciso: deshabilitar el modo autocommit, luego ejecutar las instrucciones SQL, y terminar con commit si todo va bien o rollback en otro caso.



# Transacciones.

- Métodos frecuentemente usados:

`void setAutoCommit(boolean b)`

Define modo de autocompromiso de la conexión (por omisión es true).

`boolean getAutoCommit()`

Obtiene modo de autocompromiso.

`void commit()`

Compromete todas las sentencias desde último compromiso.

`void rollback()`

Deshace los efectos de las sentencias SQL desde último compromiso.



# Niveles de aislamiento transaccional.

- La interface `java.sql.Connection`, opera con los siguientes niveles:
  - **TRANSACTION\_NONE**: Sin soporte transaccional.
  - **TRANSACTION\_READ\_COMMITTED**: Permite lecturas solo sobre datos comprometidos. Es el nivel por defecto en JDBC.
  - **TRANSACTION\_READ\_UNCOMMITTED**: Permite lecturas sobre datos no comprometidos.
  - **TRANSACTION\_REPEATABLE\_READ**: Bloquea los datos leídos.
  - **TRANSACTION\_SERIALIZABLE**: Solo una transacción al mismo tiempo, elimina todos los problemas de concurrencia pero como realiza muchos bloqueos afecta el rendimiento y también disminuyen los accesos concurrentes.

NOTA: No todos los drivers/BDs tienen que soportar todos los niveles de aislamiento. Por lo regular soportan **TRANSACTION\_READ\_COMMITTED** que es el nivel por defecto y **TRANSACTION\_SERIALIZABLE**.



# Excepciones JDBC.

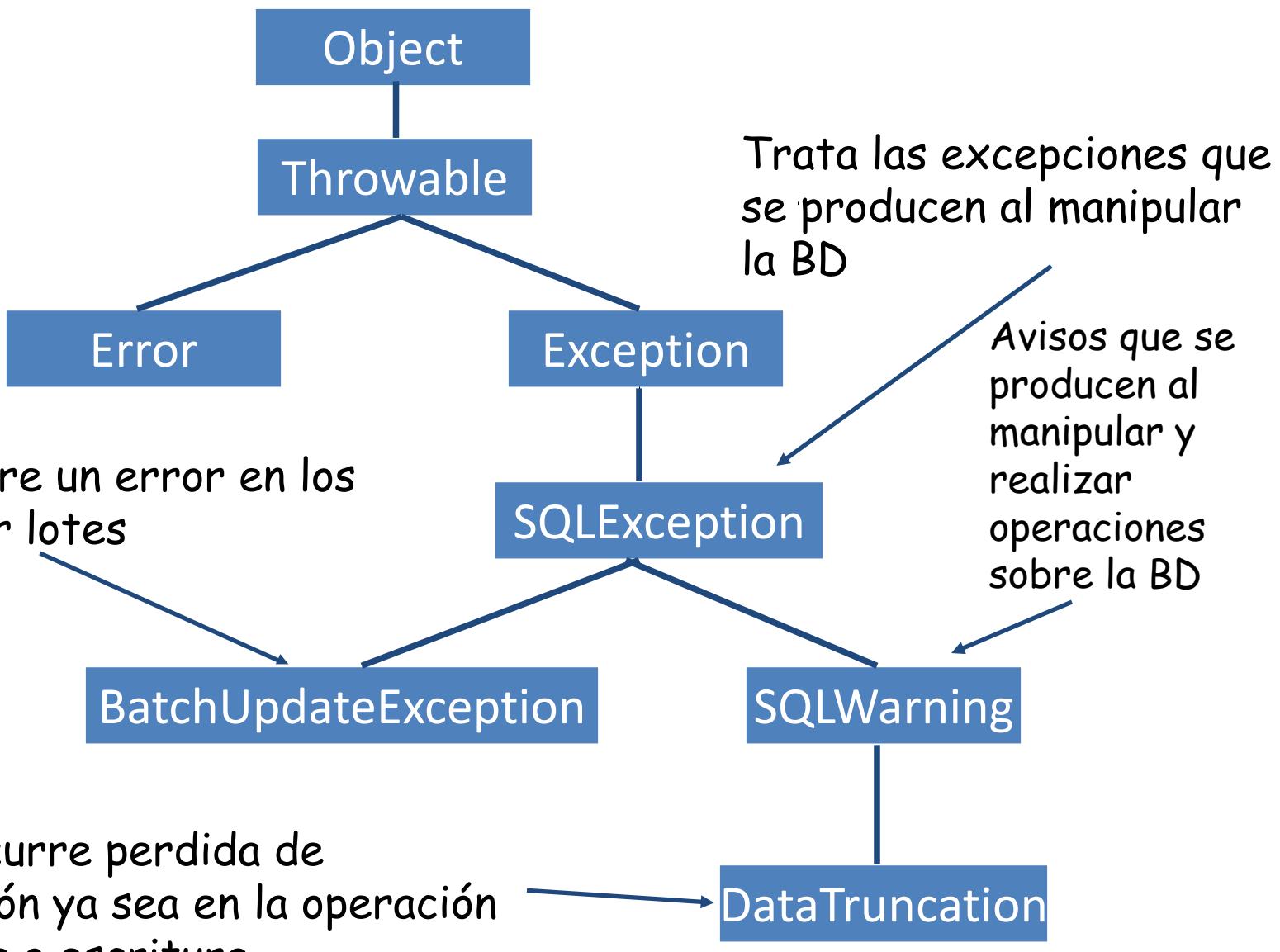
- La mayor parte de las operaciones que nos proporciona el API JDBC lanzarán la excepción `java.sql.SQLException` en caso de que se produzca algún error a la base de datos (por ejemplo: errores en la conexión, sentencias SQL incorrectas, falta de privilegios, etc.).
- Por este motivo es necesario dar un tratamiento adecuado a estas excepciones y encerrar todo el código JDBC entre bloques `try/catch`. Por ejemplo:

```
try {  
}  
  
catch(SQLException sqle) {  
    System.out.println("SQLException:" + sqle.getMessage());  
    sqle.printStackTrace();  
}
```

Información al usuario de los errores  
que se han producido.



# Excepciones JDBC.



# Modulo 04

# Práctica de Laboratorio.



# Objetivos.

- Después de completar este modulo, el participante será capaz de:
  - Escribir código Java para crear procedimientos almacenados.
  - Escribir código Java para utilizar procedimientos almacenados.
  - Explicar como funciona el manejo de las transacciones.



# Práctica de Laboratorio.

- **Paso 1.** Crear una base de datos denominada Inventario y crear las tablas en base a las definiciones siguientes:

Producto

Productoid	INT	4
NombreProducto	VARCHAR	200
PrecioProducto	FLOAT	8
ColorProducto	CHAR	10

Proveedor

ProveedorId	INT	4
NombreProveedor	VARCHAR	100
RegionProveedor	CHAR	50
DescripcionProveedor	CHAR	250

Producto\_Proveedor

Productoid	INT	4
ProveedorId	INT	4
Cantidad	INT	4



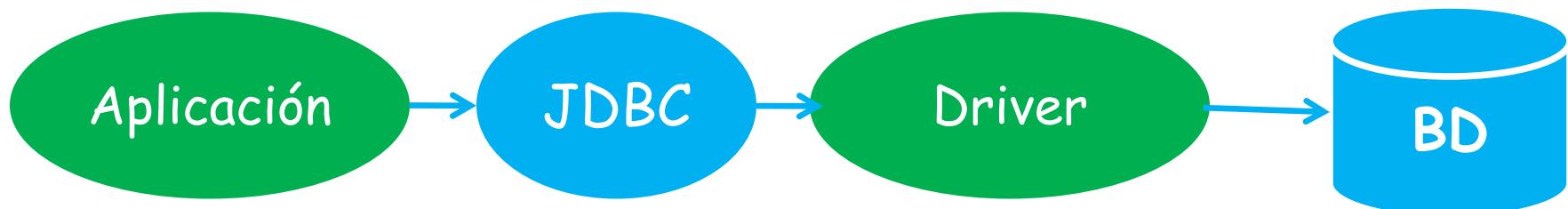
# Práctica de Laboratorio.

- **Paso 2.** Escribir un programa en Java que cree un procedimiento almacenado para recuperar todos los datos de la tabla Producto\_Proveedores dado el id de un proveedor.
- **Paso 3.** Escribir código Java que llame a ese procedimiento almacenado y despliegue los datos utilizando un ResultSet.
- **Paso 4.** Escribir un programa completo en Java que utilice sentencias precompiladas (prepareStatements) para las operaciones de la Base de Datos.



# Introducción.

- JDBC es una API de Java para ejecutar sentencias SQL; es una especificación formada por un conjunto de clases abstractas e interfaces que deben implementar todos los fabricantes del controlador JDBC.
- El driver desarrollado por el fabricante es el que hace de intermediario entre JDBC y las aplicaciones que acceden a las bases de datos.



Nota: JDBC es una marca registrada y no un acrónimo. Sin embargo a menudo se le menciona como: Java Database Connectivity.



# Introducción.

- La idea principal de la API de JDBC es que el programador codifique sus programas abstrayéndose de los detalles de implementación y configuración del Sistema Manejador de Base de Datos.
- Con la API de JDBC el programador puede:
  - Manipular las conexión a las Bases de Datos.
  - Manejar transacciones.
  - Utilizar sentencias precompiladas.
  - Realizar llamadas a procedimientos almacenados.
  - Acceder al diccionario de datos (Metadatos), etc.
- La API de JDBC está definida en los paquetes **java.sql** y **javax.sql** que vienen distribuidas en las versiones JSE y JEE respectivamente.



# Evolución de JDBC.

Versión	Incorporado en	Nombre del paquete	Contenido
JDBC 1.0	JDK 1.1	java.sql	Conectividad básica a los clientes de Base de Datos
JDBC 2.0 Core API	JDK 1.2 y posterior	java.sql	Navegación por los ResultSets, actualizaciones en Batch, soporte SQL-3
JDBC 2.0 Optional API	J2EE y posteriores	javax.sql	Diseñado para sacar provecho de los Java Beans
JDBC 2.1 Optional API	No incorporado	javax.sql	Mejora incremental e incorpora nuevas funcionalidades
JDBC 3.0 Core API	JDK 1.4 y posteriores	java.sql	Connection y sentencias de tipo Pooling entre otras características
JDBC 4.0	JDK 1.6 y posteriores	java.sql	Nueva interfaz Wrapper, Carga automática del driver JDBC, mejoras en las conexiones, etc



# Creación de la base de datos.

```
DROP DATABASE IF EXISTS AGENDA;  
CREATE DATABASE AGENDA;  
USE AGENDA;  
  
CREATE TABLE IF NOT EXISTS contactos(  
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    nombre VARCHAR(80) NOT NULL,  
    telefono VARCHAR(20) NULL,  
    email VARCHAR(60) NULL  
);  
  
GRANT SELECT,INSERT,UPDATE,DELETE  
ON AGENDA.* TO 'curso'@'localhost'  
IDENTIFIED BY 'curso';
```



# Conexión a JDBC.

- Una vez que tenemos creada la Base de Datos AGENDA con su respectiva tabla contactos el siguiente paso es establecer una conexión. Para ello tenemos dos vertientes:
  - Conexión directa:
    - Esta alternativa establece una conexión directa a la base de datos y solo se recomienda si no existe otra opción.
  - Conexión con datasource:
    - Abstacta los detalles de acceso (carga los drivers, login, etc.)
    - Suelen mantener un pool de conexiones.
    - Proporcionado por el contenedor utilizando el API JNDI (Java Naming and Directory Interface).
    - Los detalles de acceso se indican en archivos de configuración.



# Conexión a JDBC.

- En este curso utilizaremos la conexión directa para trabajar con JDBC.
- Las siguientes instrucciones son válidas para el IDE Eclipse:
- Es necesario añadir el JAR del driver a nuestro proyecto en eclipse, para eso:
  - ⌘Menú Project / Properties / Java Build Path.
  - ⌘Add external Jars.
- Esto tendríamos que realizarlo para cada proyecto.
  - ⌘Menú Window / Preferences / Java / Installed JREs / Edit / Add
  - External Jars
  - ⌘Añadimos el jar.
- ⌘A partir de ahora, tendremos la librería incluida para cualquier proyecto.



# Conexión a JDBC.

- Una vez que tenemos creada la Base de Datos AGENDA con su respectiva tabla contactos el siguiente paso es establecer una conexión. Para ello tenemos dos vertientes:
  - Conexión directa:
    - Esta alternativa establece una conexión directa a la base de datos y solo se recomienda si no existe otra opción.
  - Conexión con datasource:
    - Abstacta los detalles de acceso (carga los drivers, login, etc.)
    - Suelen mantener un pool de conexiones.
    - Proporcionado por el contenedor utilizando el API JNDI (Java Naming and Directory Interface).
    - Los detalles de acceso se indican en archivos de configuración.



# Cargar el controlador JDBC.

- Algunos drivers populares para otras Bases de Datos:

<b>SMBD</b>	<b><i>Nombre del Driver JDBC</i></b>
MySQL	Driver: com.mysql.jdbc.Driver Formato url: jdbc://hostname/databaseName
Oracle	Driver: oracle.jdbc.driver.OracleDriver Formato url: jdbc:oracle:thin@hostname:portnumber:databaseName
DB2	Driver: com.ibm.db2.jdbc.net.DB2Driver Formato url: jdbc:db2:hostname:portnumber/databaseName
Access	Driver: com.jdbc.odbc.jdbc.OdbcDriver Formato url: jdbc.odbc.databaseName



# Cargar el controlador JDBC.

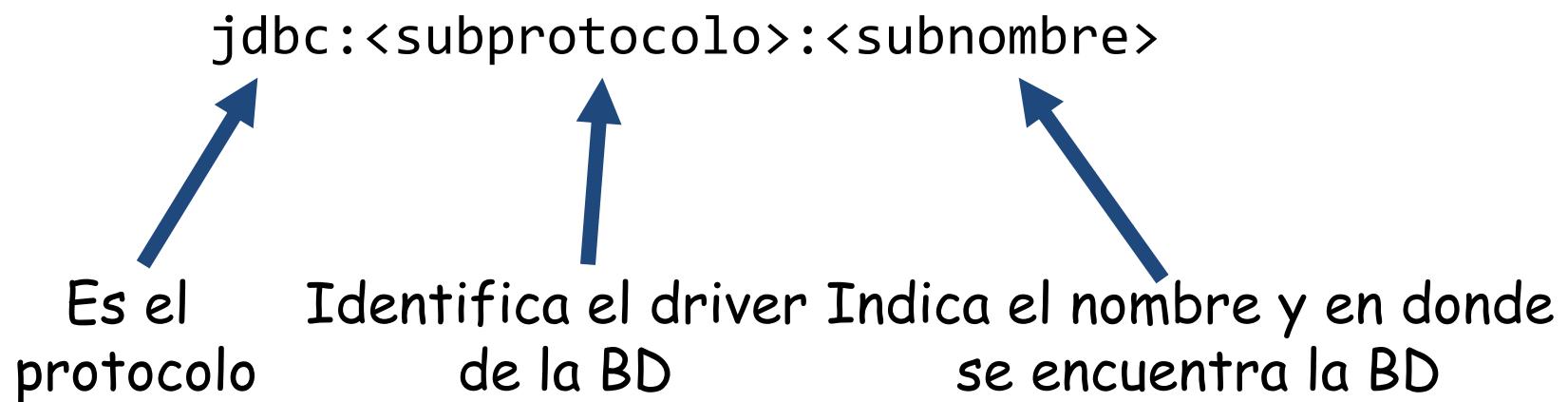
- Es importante mencionar que la clase `java.sql.DriverManager` trabaja entre el usuario y los controladores. Tiene en cuenta los drivers disponibles y a partir de ellos establece una conexión entre una Base de Datos y el controlador adecuado para esa Base de Datos.
- Además de esta labor principal, el `DriverManager` se ocupa de mostrar mensajes de log del driver y también el tiempo límite en espera de conexión del controlador (time out).
- Normalmente, en un programa Java el único método que un programador deberá utilizar de la clase `DriverManager` es `getConnection()` y como su nombre lo indica establece una conexión con una Base de Datos.



# Establecer una conexión JDBC.

- Una vez registrado el controlador con el DriverManager, el método getConnection() debe especificar la fuente de datos a la que se desea acceder.
- En JDBC, una fuente de datos se especifica por medio de un URL con el prefijo del protocolo jdbc:, la sintaxis y la estructura del protocolo es la siguiente:

jdbc:<subprotocolo>:<subnombre>



Es el protocolo      Identifica el driver de la BD      Indica el nombre y en donde se encuentra la BD



# Establecer una conexión JDBC.

- La documentación de la Base de Datos establece el tipo de controlador a utilizar.
- El contenido y la sintaxis de subnombre dependen del subprotocolo, pero en general indican el nombre y la ubicación de la fuente de datos.
- Cuando la URL de la Base de Datos se carga correctamente se obtiene una conexión abierta que se puede utilizar para crear sentencias JDBC que pasen nuestras sentencias SQL al controlador de la Base de Datos.
- Si no logra establecerse una conexión entonces se lanza una SQLException.



# Establecer una conexión JDBC.

- Para nuestros ejemplos, la URL de conexión es la siguiente:

`jdbc:mysql://localhost/Agenda`

- Las instrucciones típicas que utilizaremos en nuestros ejemplos es la siguiente:

```
String url = "jdbc:mysql://localhost/Agenda";
String login = "curso";
String password = "curso";
Connection conn =
DriverManager.getConnection(url,login,password);
```



## TestConexion.java

```
import java.sql.*; ← Importamos la API JDBC  
...  
try {  
  
    Class.forName("com.mysql.jdbc.Driver").newInstance();  
    Connection conn =  
    DriverManager.getConnection(url,login,password);  
    ...  
    conn.close(); ← Cierra la conexión  
}  
catch {  
...  
} ← Si no se logra cargar el driver o sucede una  
     excepción utilizando JDBC se lanza la  
     excepción correspondiente
```

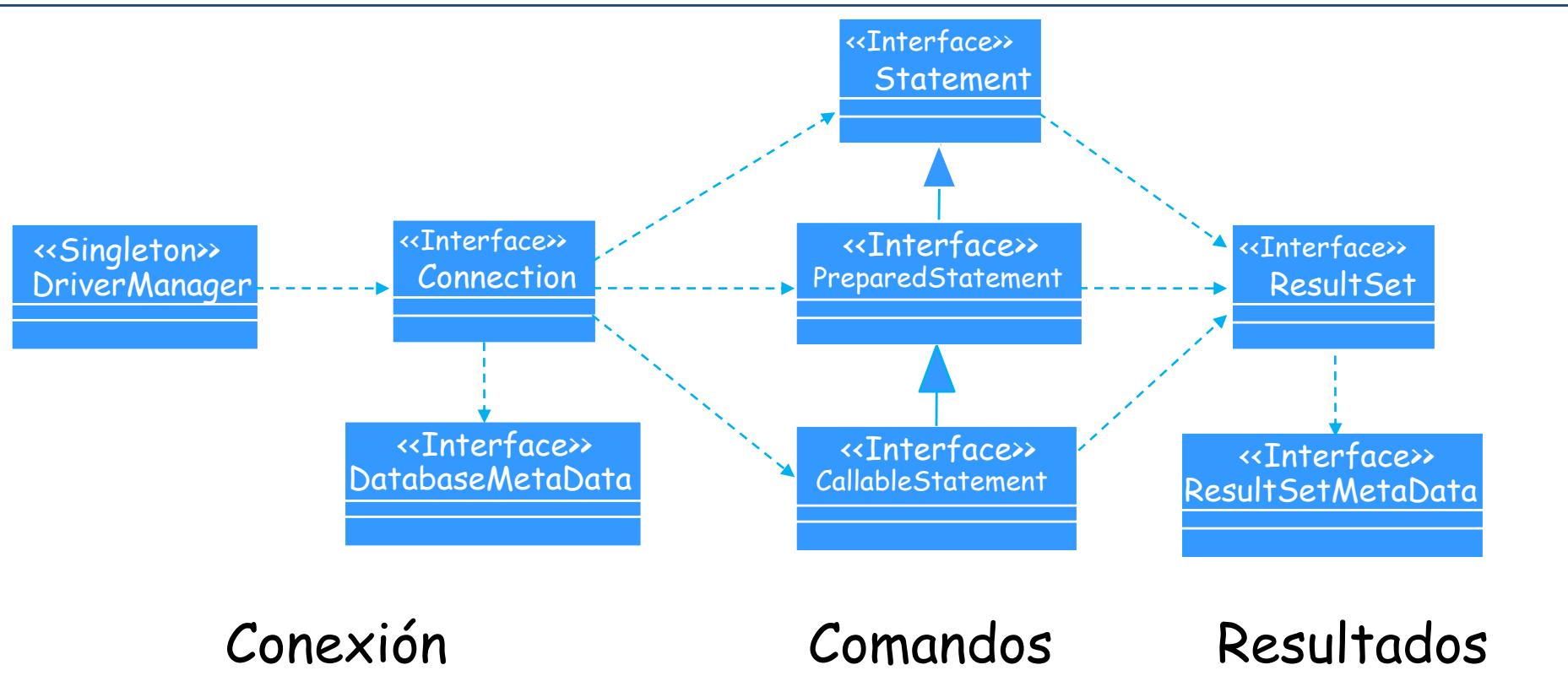
Establece el Driver a utilizar

Crea la conexión



# Arquitectura del API JDBC.

- Arquitectura: para acceder a una base de datos es necesario un driver, la implementación de todas las interfaces de la API.



# Creación de sentencias.

- Ya que establecimos la conexión con la Base de Datos podemos realizar lo siguiente:
  - Crear objetos de tipo Statement, PreparedStatement y CallableStatement
  - Obtener información con los objetos DatabaseMetadata
  - Controlar las transacciones vía commit() y rollback()
  - Delimitar el nivel al que queremos manejar las transacciones
- El siguiente diagrama ilustra el uso de las instrucciones más importantes para trabajar con JDBC.



# Objetos Statement.

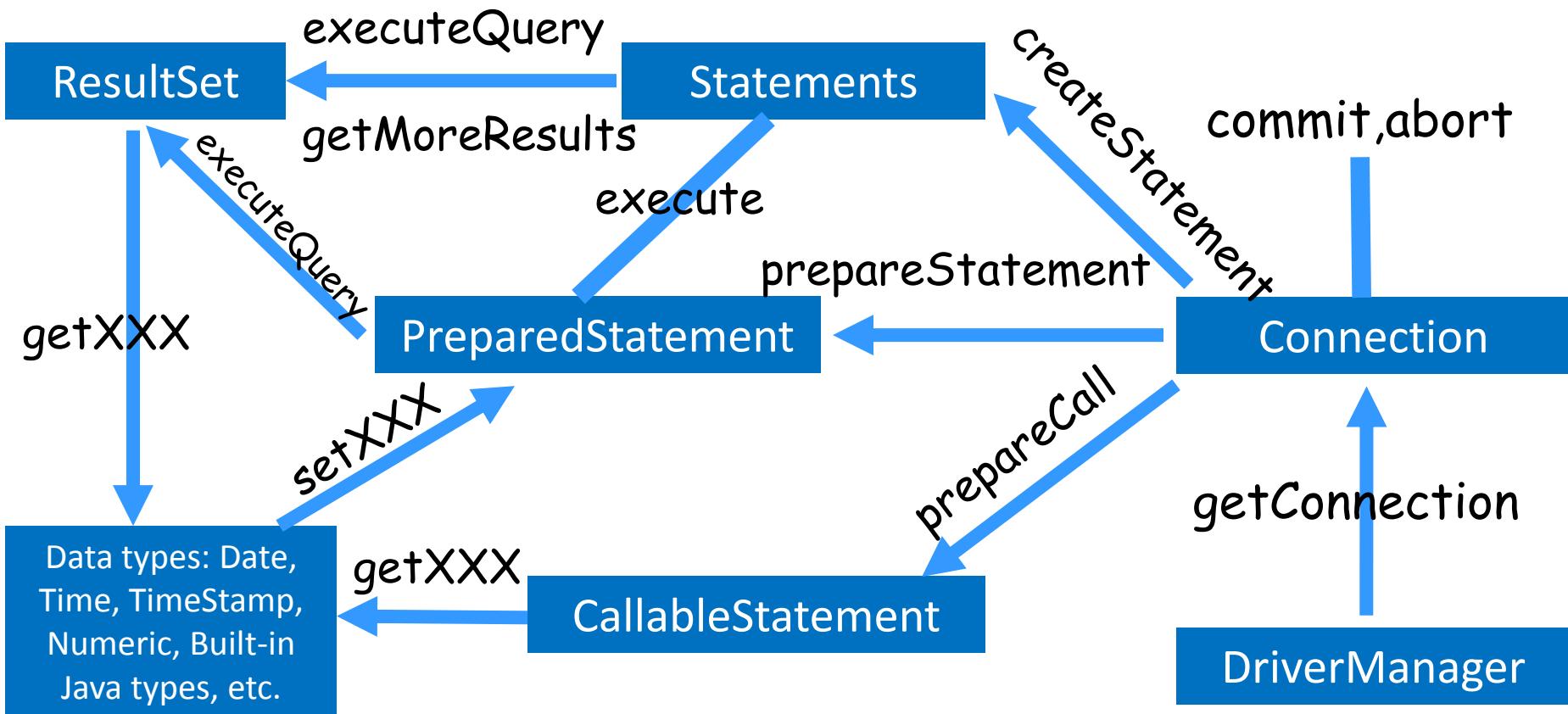
- Los objetos de tipo Statement son creados a partir de una conexión existente utilizando el método `createStatement()`. La sintaxis es:

```
Statement stmt = conn.createStatement();
```

- Los objetos Statement envían comandos SQL a la Base de Datos .
- Para una sentencia SELECT, el método a ejecutar es `executeQuery()`. Para sentencias que crean o modifican tablas, el método a utilizar es `executeUpdate()`. Para sentencias que no sabemos el tipo de retorno utilizaremos el método `execute()`.



# Creación de sentencias.



# ResultSetMetaData.

- Algunos métodos para ResultSetMetaData:

Nombre del método	Valor R	Descripción
getColumnName()	int	Devuelve el número de columnas que contiene la tabla de resultados
getColumnLabel(int column)	String	Devuelve la etiqueta sugerida para la columna
getColumnName(int column)	String	Devuelve el nombre de la columna
getColumnType(int column)	String	Devuelve el tipo de dato SQL que contiene la columna
getColumnTypeName(int column)	String	Devuelve el nombre del tipo de dato que contiene la columna específicos del SMBD
getTableName(int column)	String	Devuelve el nombre de la tabla a la que pertenece la columna



# ¿Qué necesitamos para trabajar con JDBC?.



- Para trabajar con JDBC y en consecuencia con las bases de datos en Java necesitamos configurar el ambiente de trabajo.

1

Instalar JDK.

Descargar la última versión de la plataforma Java que ya contiene los paquetes `java.sql` y `javax.sql` y seguir las instrucciones de instalación.

<http://java.sun.com/javase/index.jsp>

2

Instalar el driver.

Dependiendo del fabricante de el Sistema Manejador de Base de Datos vamos a necesitar un **driver para JDBC**. La instalación del mismo consiste en copiar el driver en nuestra computadora en una ubicación específica.

<http://developers.sun.com/products/jdbc/drivers>

3

Instalar el SMBD.

Si no hay una Base de Datos instalada en su computadora busque alguna (MS Access, MySQL, MS SQL Server, Oracle, Java DB, etc) y siga las instrucciones del fabricante.



# ¿Qué necesitamos para trabajar con JDBC?.



- Para el desarrollo de nuestro curso vamos a utilizar la **IDE de NetBeans** que ya trae todo integrado incluyendo el **SMBD Java DB**.

<http://www.netbeans.org/downloads/index.html>

Descargar NetBeans IDE 6.7.1

6.7.1 | 6.8 M1 | Python EA2 | Desarrollo | Archivo

Correo electrónico (opcional):  Suscribirse a noticias:  Mensualmente  Semanalmente  Contactarme a esta dirección Idioma del IDE: English Plataforma: Windows 2000/XP/Vista Nota: El paquete UML está disponible en el Centro de Actualización del IDE

Tecnologías *	Java SE	JavaFX	Java	Ruby	C/C++	PHP	All
Java SE	•	•	•				•
JavaFX		•					•
Java Web y EE							•
Java ME							•
Ruby							•
C/C++							•
Groovy							•
PHP							•
Servidores incluidos			•				
Sun GlassFish Enterprise Server v2.1				•			
Sun GlassFish Enterprise Server v3 Prelude			•		•		
Apache Tomcat 6.0.18			•				

Paquetes de descarga de NetBeans IDE

Descargar la última versión y seguir las instrucciones de instalación.

Descargar

Download Libre, 47 MB Download Libre, 90 MB Download Libre, 238 MB Download Libre, 59 MB Download Libre, 30 MB Download Libre, 26 MB Download Libre, 302 MB



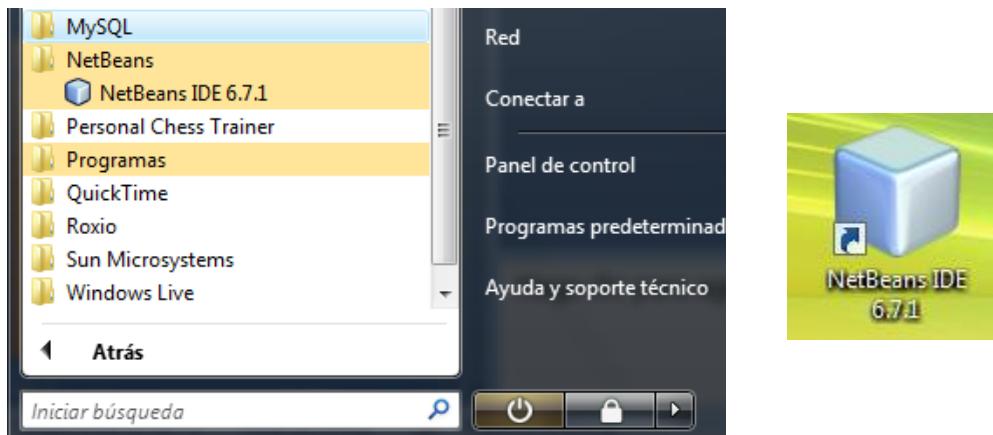
# Configuración de la aplicación Java.

- Una vez que hemos instalado el ambiente de trabajo (IDE NetBeans), lo que tenemos que hacer es configurar nuestro proyecto Java para poder realizar los ejemplos que vamos a desarrollar a lo largo de nuestro curso. Para ello vamos a seguir los siguientes pasos:

1

Arrancar  
NetBeans.

Doble clic sobre el ícono que está en el escritorio o bien buscar la aplicación en el menú Inicio.





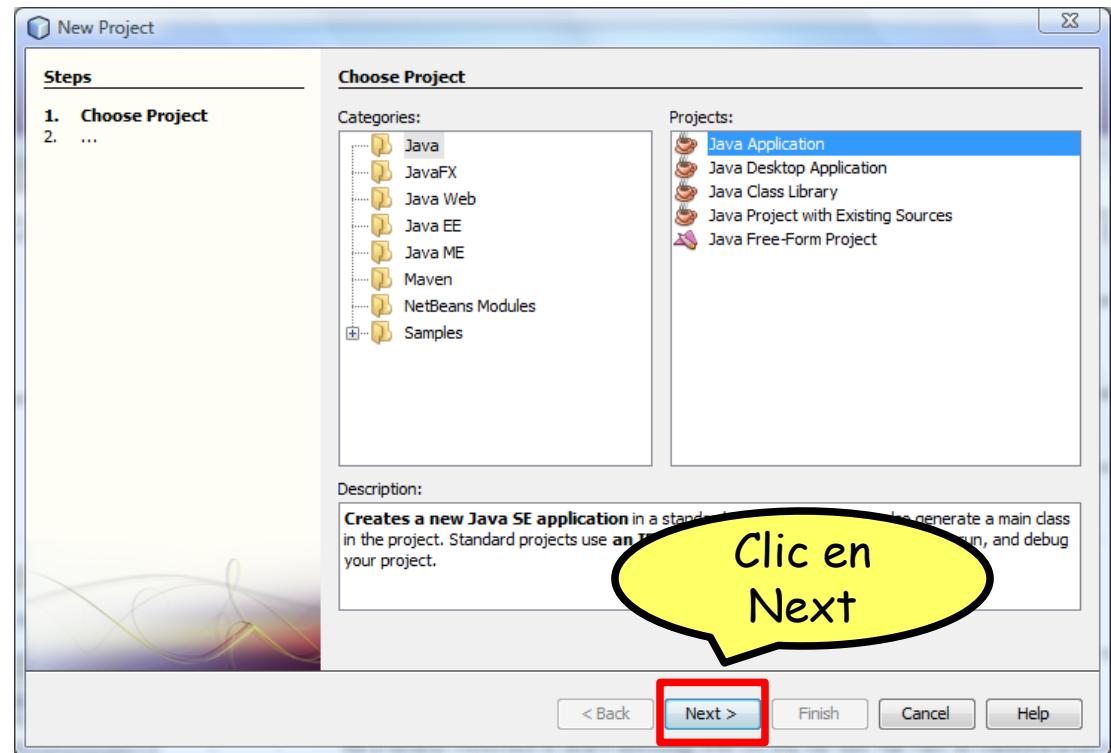
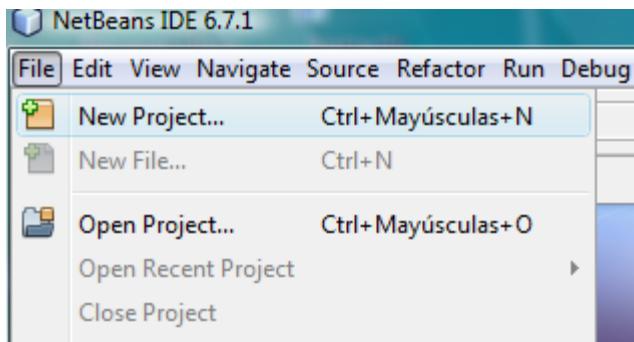
# Configuración de la aplicación Java.

2

Crear el  
proyecto Java

Con la IDE en ejecución, el siguiente paso es crear el proyecto Java. Selecciona el Menú File>New Project.

Aparecerá un asistente. Seleccionar la categoría Java y el tipo de proyecto será Java Application. Clic en el botón Next>.





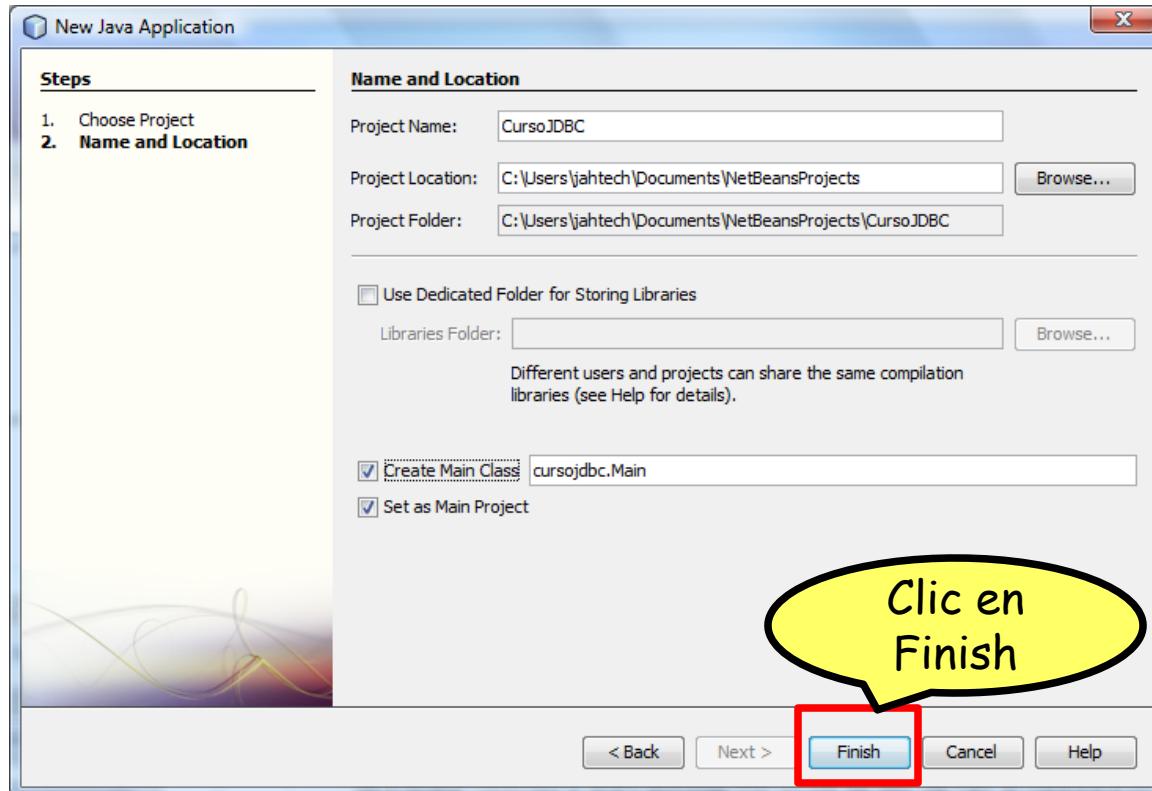
# Configuración de la aplicación Java.

3

Nombre y  
ubicación del  
proyecto

El siguiente paso es proporcionar el nombre y ubicación del proyecto. El nombre del proyecto será: *CursoJDBC* y la ubicación será la ruta asignada por default.

Clic en el botón *Finish*.





# Configuración de la aplicación Java.

3

Nombre y  
ubicación del  
proyecto

Finalmente hemos configurado nuestro proyecto Java y estamos listos para empezar a trabajar con los códigos de ejemplo de este curso.

```
/*  
 * To change this template, choose Tools | Templates  
 * and open the temp  
 */  
  
package cursojdbc;
```

¡Estoy listo  
para  
programar!



# Creación de la base de datos CoffeeBreak.



- Ahora necesitamos definir nuestra base de datos para el curso que llamaremos: **CoffeeBreak**.
- Suponga que el propietario de un café llamado “Coffee Break” utiliza una base de datos para controlar la cantidad de tazas de café que se venden en una semana determinada.
- En un enfoque minimalista el propietario solo utiliza dos tablas: una para los **tipos de café** y otra para los **proveedores**.



# Creación de la base de datos CoffeeBreak.

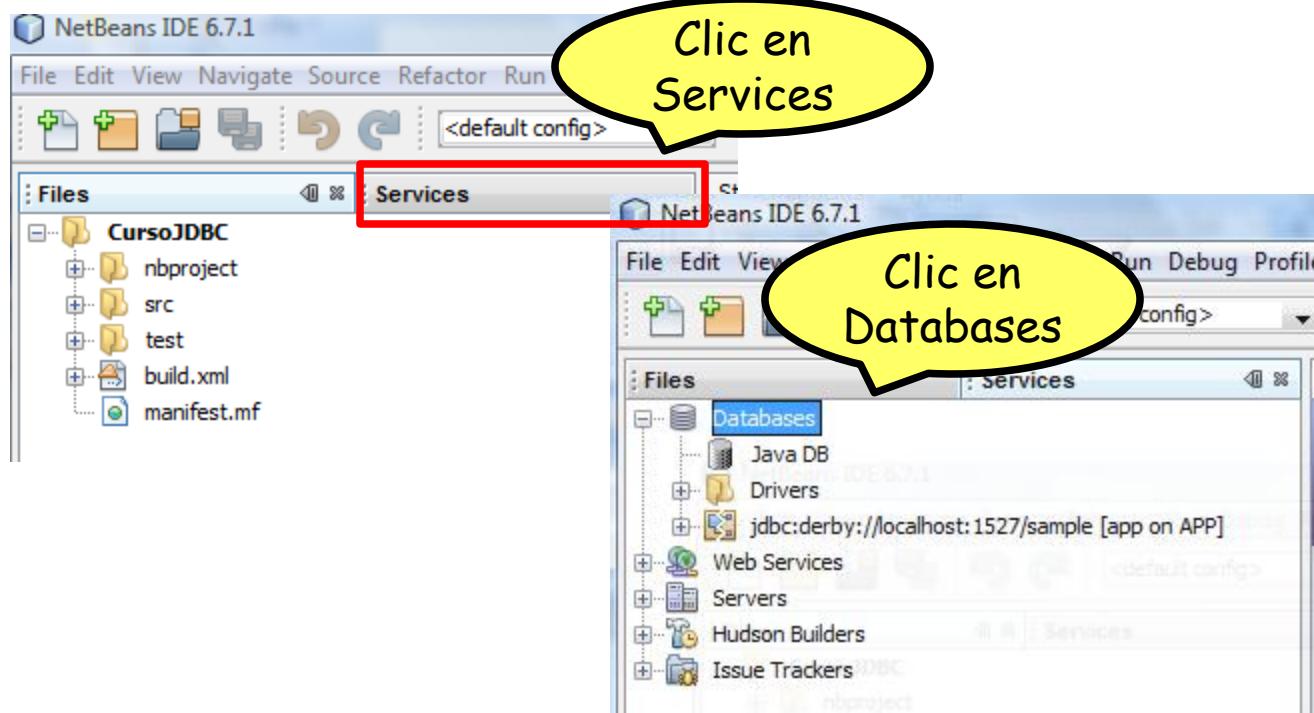


- En nuestro proyecto Java CursoJDBC de NetBeans realizar lo siguiente:

1

Localizar y clic en la pestaña Services.

En nuestro proyecto *CursoJDBC* localizar la pestaña **Services** y dar un clic con el ratón. Luego seleccionar la opción **Databases**.



# Creación de la base de datos CoffeeBreak.

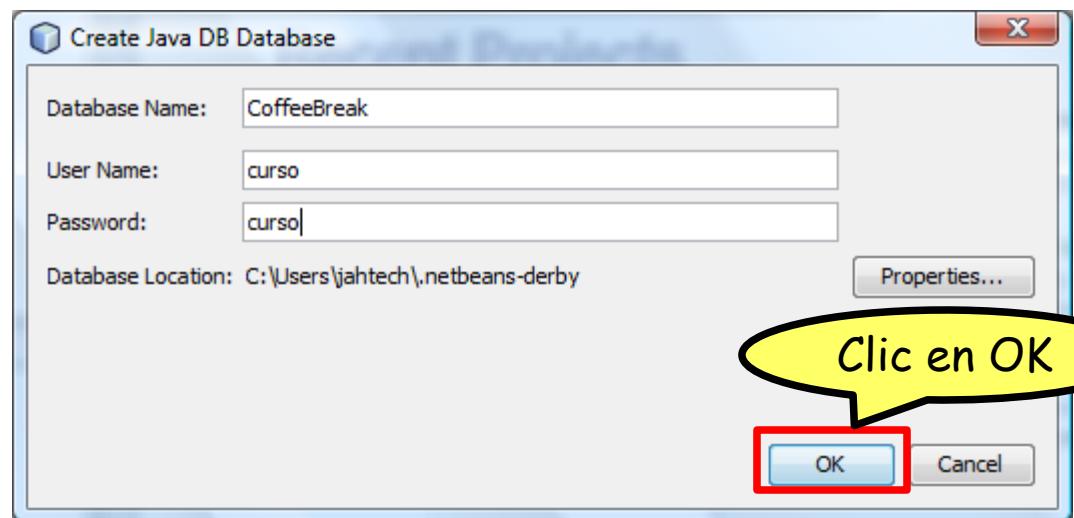
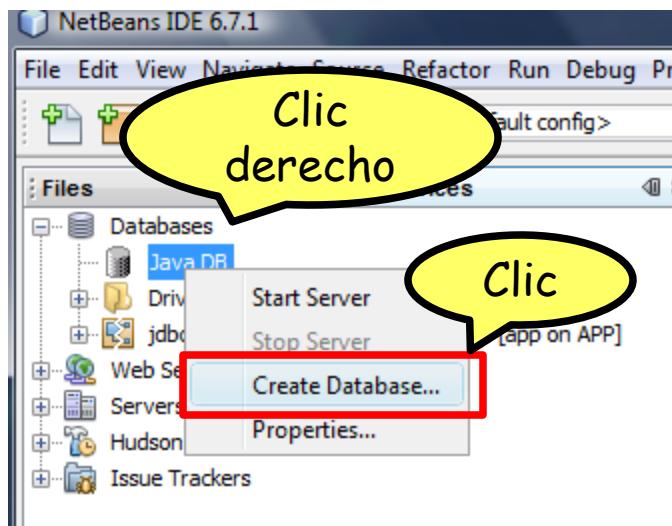


- Ahora vamos a crear la Base de Datos utilizando el SMBD Java DB para ello:

2

Clic derecho del ratón en Java DB.

En el menú contextual seleccionar la opción: **Create Database**. Después en el asistente proporcionar el nombre de la base de datos: **CoffeeBreak** con el nombre de usuario y password: **curso**. Clic en el botón **OK**.



# Creación de la base de datos CoffeeBreak.

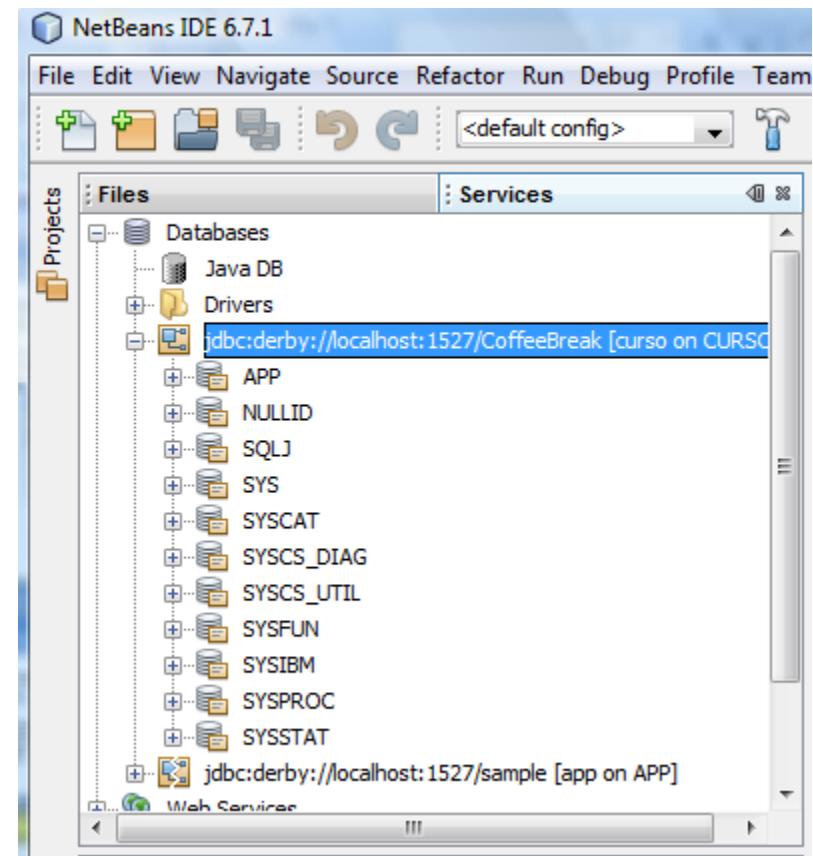
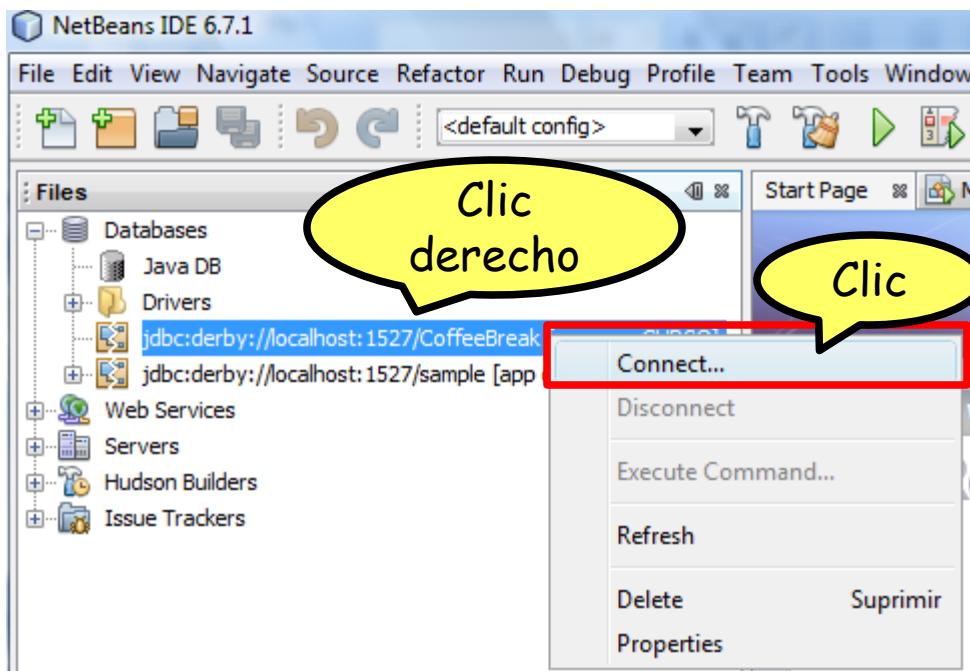


- El siguiente paso es conectarse con la base de datos JavaDB:

2

Seleccionar  
la BD  
CoffeeBreak.

En la pestaña Services y JavaDB localizar la base de datos CoffeeBreak. Clic derecho y seleccionar la opción Connect.



# Creación de la base de datos CoffeeBreak.

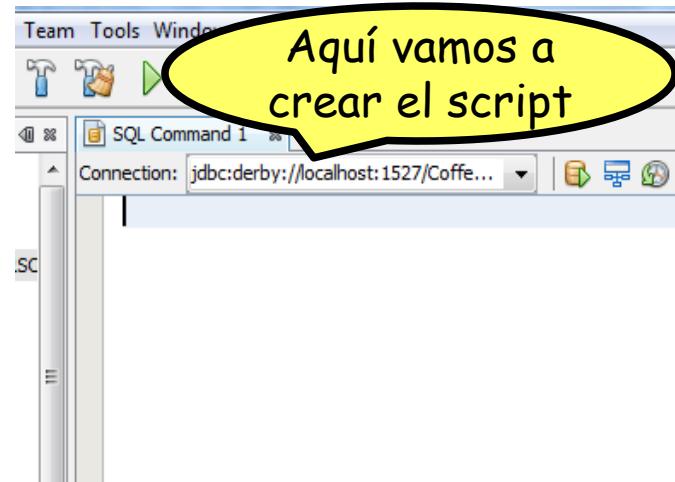
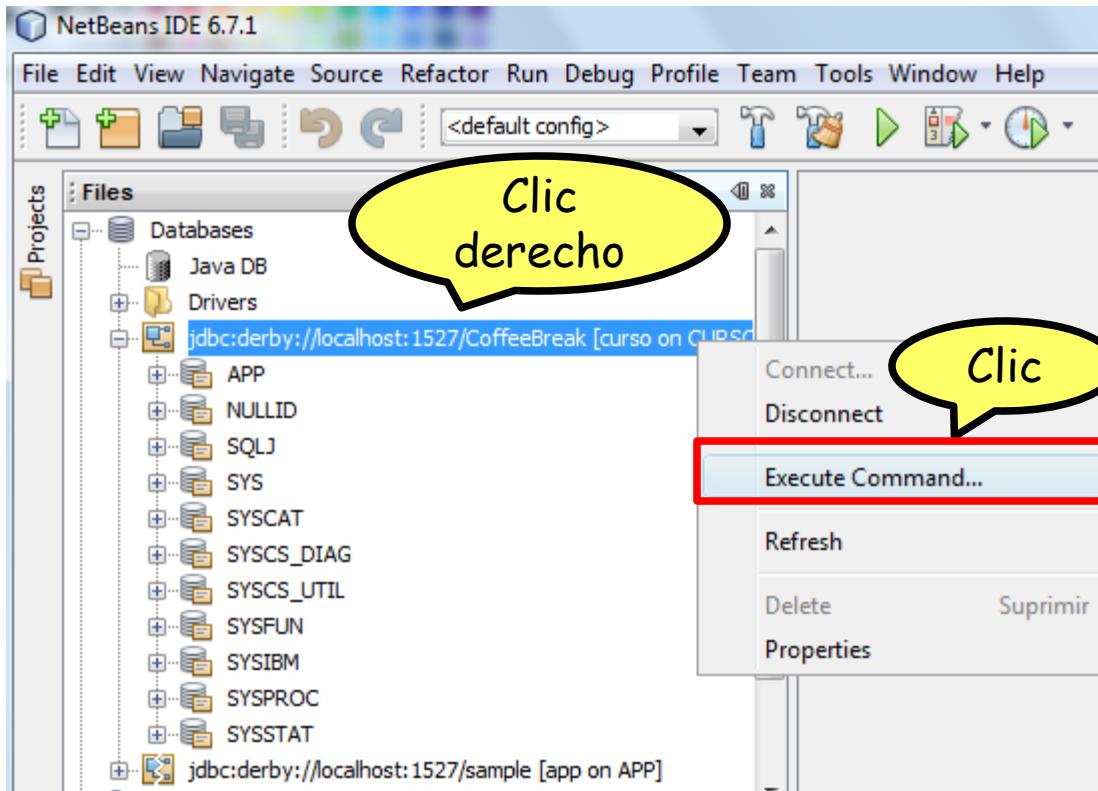


- Ahora estamos listos para crear las tablas de la base de datos:

2

Seleccionar  
la BD  
CoffeeBreak.

Clic derecho en la BD CoffeeBreak y seleccionar la opción Execute Command.



# Creación de la base de datos CoffeeBreak.



3

Crear el  
Script SQL.

```
CREATE TABLE PROVEEDOR (
    PROV_ID INTEGER NOT NULL,
    PROV_NOMBRE VARCHAR(40),
    CALLE VARCHAR(40),
    CIUDAD VARCHAR(20),
    ESTADO VARCHAR(20),
    CODIGO_POSTAL VARCHAR(5),
    PRIMARY KEY(PROV_ID)
);
CREATE TABLE CAFE(
    CAFE_NOMBRE VARCHAR(32) NOT NULL,
    PROV_ID INTEGER,
    PRECIO FLOAT,
    VENTAS INTEGER,
    TOTAL INTEGER,
    PRIMARY KEY(CAFE_NOMBRE),
    CONSTRAINT CAFE_FK1 FOREIGN
    KEY(PROV_ID)
    REFERENCES PROVEEDOR(PROV_ID)
);
```

En el editor de código escribir el siguiente código SQL.

4

Clic en el  
botón Run  
SQL.

Por último damos clic en  
el botón Run SQL para  
ejecutar el script.

The screenshot shows the MySQL Workbench interface. On the left, the 'Files' pane displays the database structure under 'Databases'. A red box highlights the 'CURSO' schema, which contains 'Tables' (CAFE and PROVEEDOR), 'Views', and 'Procedures'. A yellow speech bubble labeled 'Tablas' points to the 'Tables' folder. On the right, the 'SQL Command 1' window shows the SQL code for creating the 'PROVEEDOR' and 'CAFE' tables. A yellow speech bubble labeled 'Clic' points to the 'Run SQL' button at the top of the window. The connection is set to 'jdbc:derby://localhost:1527/CoffeeBreak [curso on CURSO]'.

```
CREATE TABLE PROVEEDOR (
    PROV_ID INTEGER NOT NULL,
    PROV_NOMBRE VARCHAR(40),
    CALLE VARCHAR(40),
    CIUDAD VARCHAR(20),
    ESTADO VARCHAR(20),
    CODIGO_POSTAL VARCHAR(5),
    PRIMARY KEY(PROV_ID)
);
CREATE TABLE CAFE(
    CAFE_NOMBRE VARCHAR(32) NOT NULL,
    PROV_ID INTEGER,
    PRECIO FLOAT,
    VENTAS INTEGER,
    TOTAL INTEGER,
    PRIMARY KEY(CAFE_NOMBRE),
    CONSTRAINT CAFE_FK1 FOREIGN
    KEY(PROV_ID)
    REFERENCES PROVEEDOR(PROV_ID)
);
```

# Conectarse a la base de datos: JavaDB.



- En el proyecto creado anteriormente en la IDE NetBeans: CursoJDBC vamos a crear una clase de nombre **TestConexion.java** para probar la conectividad a la base de datos JavaDB.
- En el caso de que tengamos que conectarnos a una base de datos distinta tendríamos que buscar el driver JDBC en particular, instalar, configurar y consultar la documentación del fabricante.
- ¡Manos a la obra!, las siguientes diapositivas muestran paso a paso como realizar la conexión a la base de datos.





# Conectarse a la base de datos: JavaDB.

1

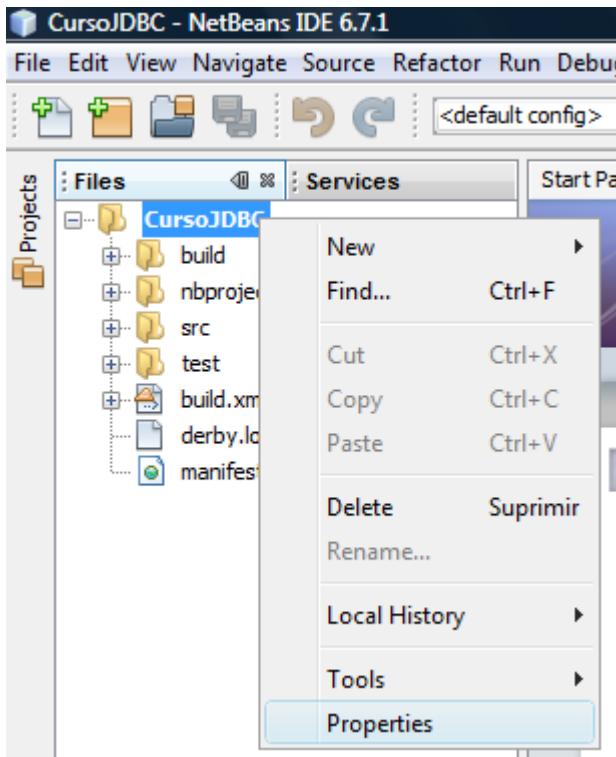
Arrancar NetBeans.

Si no está abierto la IDE NetBeans: doble clic sobre el ícono que está en el escritorio o bien buscar la aplicación en el menú Inicio.

2

Configurar el Driver JDBC.

Es necesario añadir el driver **derby.jar** para JavaDB en nuestro proyecto JavaJDBC. Clic derecho en el Proyecto CursoJDBC. Seleccionar la opción **Properties**.



Aparecerá la ventana de propiedades del proyecto (**Project Properties**). Clic en la categoría **Libraries** y luego clic en el botón **Add JAR/Folder** para buscar el driver **derby.jar** que se encuentra ubicado en la ruta **C:\Archivos de Programa\Sun\JavaDB\lib\derby.jar**

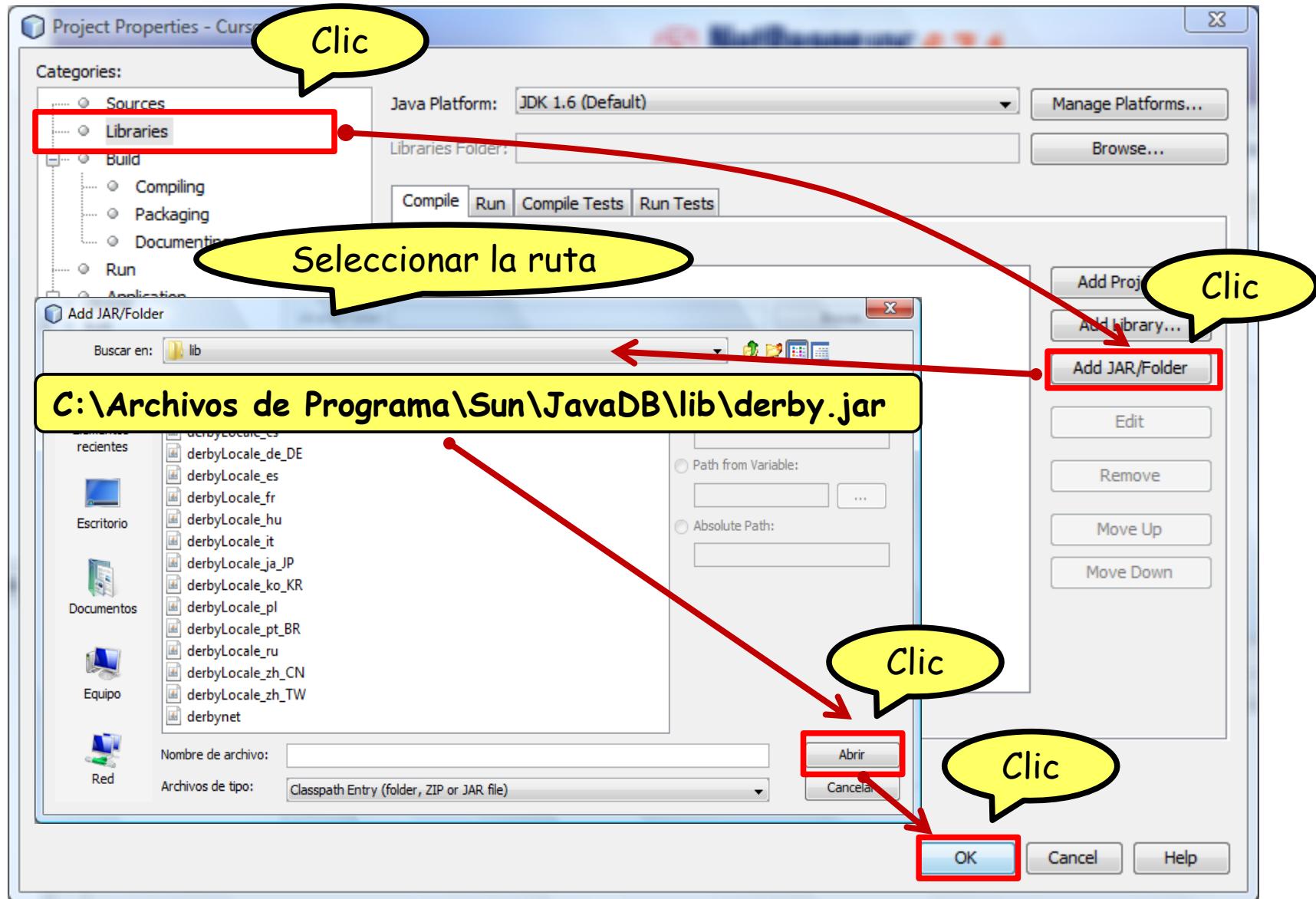
Clic en el botón **Abrir** para regresar a la ventana de propiedades del proyecto. Clic en el botón **OK**.

A partir de ahora tenemos incluida la librería para el proyecto.





# Conectarse a la base de datos: JavaDB.



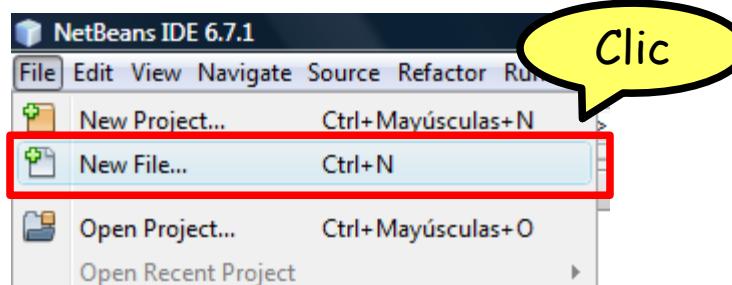


# Conectarse a la base de datos: JavaDB.

3

Crear una clase Java.

Seleccionar con un clic el menú **File>New File**. En el asistente seleccionar la categoría **Java** y el tipo de archivo **Java Class**. Luego dar clic en el botón **Next>**. Entonces proporcionar el nombre de la clase: **TestConexion** (verifique que se encuentre asociado al paquete **cursojdbc**) y clic en el botón **Finish**.

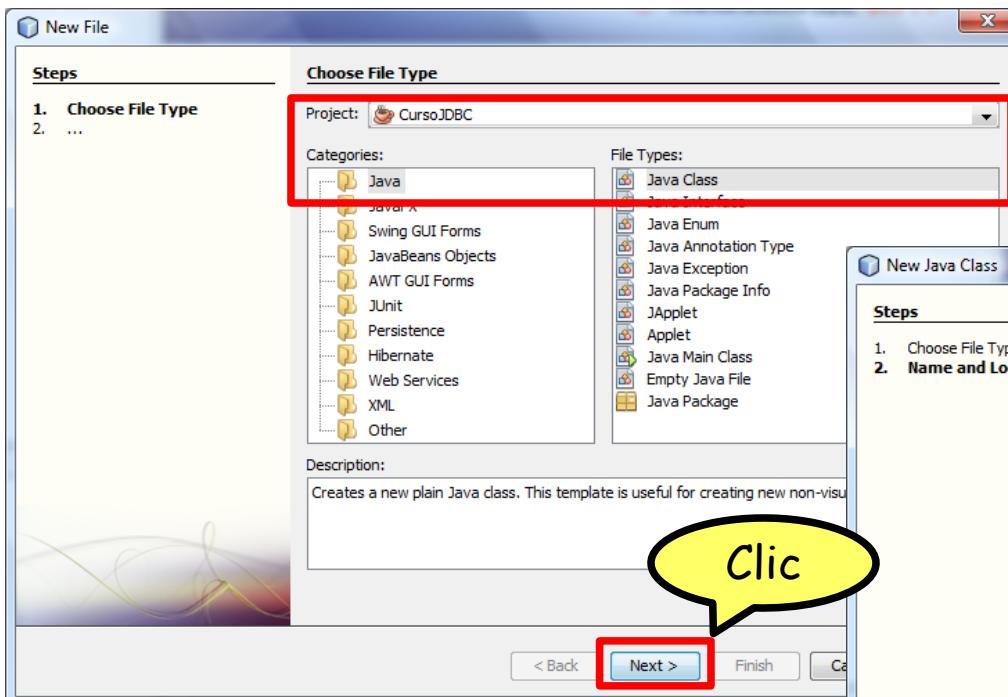


Las ilustraciones se muestran en la siguiente diapositiva.

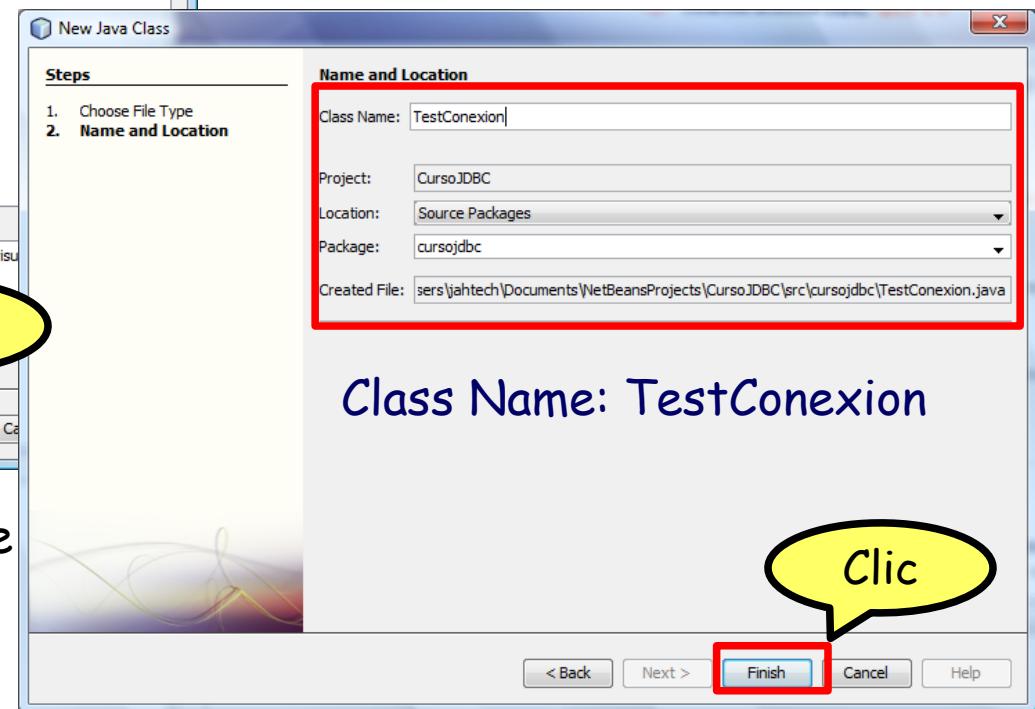




# Conectarse a la base de datos: JavaDB.



Asistente después de seleccionar  
Menú File>New File.



Class Name: TestConexion

Asistente después de  
dar clic en el botón  
**Next>**.

Una vez que damos clic en el botón **Finish** en la pantalla principal de la IDE nos aparecerá el editor de texto con la definición de la clase **TestConexion**.





# Conectarse a la base de datos: JavaDB.

4

Escribir  
método main.

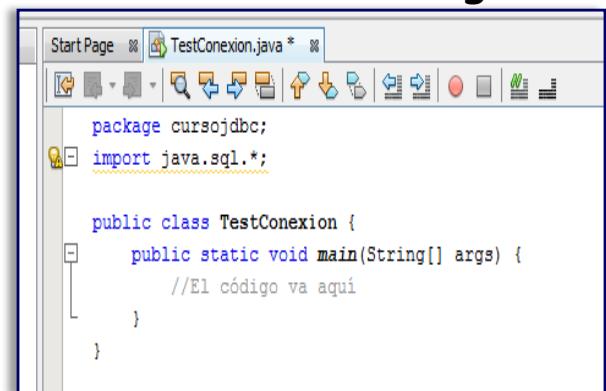
Antes de proceder a probar la conexión con la base de datos vamos a escribir en el editor de código el **método main** (y a importar la librería `java.sql`).

```
package cursojdbc;  
  
import java.sql.*;
```

El paquete `java.sql` contiene clases e interfaces para manipular bases de datos en Java.

```
public class TestConexion {  
    public static void main(String[] args) {  
        //El código va aquí  
    }  
}
```

En el editor de código.



```
Start Page × TestConexion.java *  
package cursojdbc;  
import java.sql.*;  
  
public class TestConexion {  
    public static void main(String[] args) {  
        //El código va aquí  
    }  
}
```



# Conectarse a la base de datos: JavaDB.

3

Escribir el código restante.

Cargar el driver para JavaDB es muy simple:  
`Class.forName("org.apache.derby.jdbc.Embedded");` La instrucción para conectarnos a la base de datos:  
`Connection conn =`  
`DriverManager.getConnection("jdbc:derby:CoffeeBreak;cr`  
`eate=true","curso","curso");`

The screenshot shows an IDE interface with two main windows. The left window is titled 'TestConexion.java' and contains the following Java code:

```
package cursojdbc;
import java.sql.*;

public class TestConexion {
    public static void main(String[] args) {
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            System.out.println("Driver JDBC encontrado.");
            Connection conn = DriverManager.getConnection("jdbc:derby:CoffeeBreak;create=true","curso","curso");
            System.out.println("Conexión establecida.");
        }
        catch(ClassNotFoundException cnfe) {
            System.out.println("Driver JDBC no encontrado.");
            cnfe.printStackTrace();
        }
        catch(SQLException sqle) {
            System.out.println("Error al conectarse a la BD");
            sqle.printStackTrace();
        }
    }
}
```

The right window is titled 'Output - CursoJDBC (run)' and displays the following text:

```
run:
Driver JDBC encontrado.
Conexión establecida.
BUILD SUCCESSFUL (total time: 3 seconds)
```



# Conectarse a la base de datos: JavaDB.

```
package cursojdbc;
import java.sql.*;

public class TestConexion {
    public static void main(String[] args) {
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            System.out.println("Driver JDBC encontrado.");

            Connection conn = DriverManager.getConnection
                ("jdbc:derby:CoffeeBreak;create=true","curso","curso");
            System.out.println("Conexión bd establecida.");
            conn.close();
            System.out.println("Conexión bd cerrada.");
        }
    }
}
```





# Conectarse a la base de datos: JavaDB.

```
catch(ClassNotFoundException cnfe) {  
    System.out.println("Driver JDBC no encontrado");  
    cnfe.printStackTrace();  
}  
catch(SQLException sqle) {  
    System.out.println("Error al conectarse a la BD");  
    sqle.printStackTrace();  
}  
}
```

